C FOR YOURSELF

Microsoft®

Microsoft® QuickC® Compiler

C FOR YOURSELF

VERSION 2.5

MICROSOFT CORPORATION

SbZ

Printed and bound in the United States of America.

Document No. SY10423-0290
OEM D703-2Z

10 9 8 7 6 5 4 3 2 1

Contents

Appendixes

Introduction

Ever since Microsoft introduced the QuickC® Compiler, version 1.0 in 1987, QuickC users have asked for more information on the C programming language. *C for Yourself* answers that need, particularly for those who have some programming experience but are new to the C language.

About This Book

C for Yourself assumes you have programmed before but are not familiar with C. Thus, it doesn't explain basic programming ideas such as why program loops are useful. It assumes that you already know about loops in general and now want to learn how to write them in the C language. The following list summarizes the book's contents:

- Part 1, "Learning C," covers basic C language topics such as functions and data types. The chapters in this section are designed to be read in order, from beginning to end.
- Part 2, "Using C," covers practical programming topics such as input/output and graphics. This section is organized topically, so you don't have to read the chapters in any particular order.
- Appendix A, "C Language Guide," summarizes the QuickC implementation of the C language. You can use this appendix as a quick reference once you have read Part 1 and have some familiarity with C.
- Appendix B, "C Library Guide," summarizes the most commonly used functions in the QuickC run-time library. This appendix is designed mainly for browsing when you're not using QuickC. When you are in the QuickC environment, use the Microsoft® QuickC Advisor (online help) to get information about C language features and run-time library functions.

NOTE *The pages that follow use the term "OS/2" to refer to the OS/2 systems—Microsoft® Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" refers to both the MS-DOS® and IBM® Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system.*

Using the Example Programs

The example programs in this book are available through online help. This feature allows you to load, run, and experiment with example programs as you read.

You can use online help to load and run example programs.

You must be using the QuickC environment to load an example. To load the program, select Contents from the Help menu, then select the title of this book. Find the desired program in Help, then copy it into the editor using QuickC's Copy and Paste functions.

After you copy a sample program into the QuickC Editor, you can treat it as you would any C source program. You can compile or edit the program, save it on disk, and so on.

QuickC online help includes all the significant examples in this book (it doesn't include code fragments and some very short programs). Every program that is in online help begins with a line in this general form:

```
/* POINTER.C: Demonstrate pointer basics. */
```

The line contains the program's name and a brief description of what it does. The program containing the above line is listed as POINTER.C in online help.

All the examples available in online help compile without errors at Warning Level 3, in which QuickC does the most stringent error-checking. At this Warning Level, some examples will generate the following harmless warnings:

```
C4103: 'main' : function definition used as prototype
C4035: 'main' : no return value
C4051: data conversion
```

You can eliminate these warnings by compiling at a lower Warning Level.

Programming Style Used in This Manual

The C language allows considerable flexibility in formatting source code. The style used in this book is recommended for program readability, but you do not have to use it when writing your programs. Below is a list of style guidelines used in this book for example programs:

- Each example program begins with a comment that names the program and states what it does.
- Each statement or function is listed on its own line.
- Variable and function names are in lowercase. The names of symbolic constants, such as TRUE or FALSE, are in uppercase.

- If a function doesn't take any arguments, an opening and a closing parenthesis follow the function name with no extra space:

```
getch();
```

- If a function takes arguments, a space appears after the opening parenthesis and before the closing parenthesis:

```
printf( "Number = %i", num_penguins );
```

- Binary operators such as addition and subtraction are preceded and followed by a space:

```
3 + 5
```

- If parentheses are used to control operator precedence, no extra spaces are included:

```
(3 + 5) * 2
```

- Opening and closing braces are aligned under the first character of the controlling keyword. The block underneath is indented 3 spaces:

```
if( a == b )
{
   c = 50;
   printf( "%i\n", a );
}
```

Key to Document Conventions

This book uses the following document conventions:

Example	Description
COPY TEST.OBJ C:	Uppercase letters represent DOS commands and file names.
printf	Boldface letters indicate standard features of the C language: keywords, operators, and standard library functions.
expression	Words in italics indicate placeholders for information you must supply, such as a file name. Italics are also occasionally used for emphasis in the text.

`main()`
`{`
`}`
	This typeface is used for example programs, program fragments, and the names of user-defined functions and variables. It also indicates user input and screen output.

CL *options* [[*files*...]]	A horizontal ellipsis following an item indicates that more items having the same form may follow.

`while( )`
`{`
`.`
`.`
`.`
`}`
	A vertical ellipsis tells you that part of the example program has been intentionally omitted.

SHIFT	Small capital letters denote names of keys on the keyboard. A plus sign (+) indicates a combination of keys. For example, SHIFT+F5 tells you to hold down the SHIFT key while pressing the F5 key.
"array pointer"	The first time a new term is defined, it is enclosed in quotation marks. Since some knowledge of programming is assumed, common terms such as memory or branch are not defined.
American National Standards Institute (ANSI)	The first time an acronym appears, it is spelled out.

Other Books on C Programming

This book provides an introduction to the C language and some practical programming topics. It does not attempt to teach you basic computer programming or advanced C programming techniques. The following books cover a variety of topics that you may find useful. They are listed only for your convenience. With the exception of its own publications, Microsoft does not endorse these books or recommend them over others on the same subject.

Feibel, Werner. *Advanced QuickC*, 2d ed. Berkeley, California: Osborne McGraw-Hill, 1989.

An intermediate-level C programming guide using QuickC. It includes data structures, parsing, simulations, and the DOS interface.

Hancock, Les, and Morris Krieger. *The C Primer*, 2d ed. New York: McGraw-Hill, 1986.

A beginner's guide to C programming.

Hansen, Augie. *Proficient C*. Redmond, Washington: Microsoft Press, 1987.

An intermediate-level guide to C programming.

Harbison, Samuel P., and Guy L. Steele. *C: A Reference Manual*, 2d ed. Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

A comprehensive guide to the C language and the standard library.

Hergert, Douglas. *The ABC's of QuickC*. Alameda, California: SYBEX, Inc., 1989.

A beginner's guide to QuickC programming.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, 2d ed. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

The first edition of this book is the classic definition of the C language. The second edition includes new information on the proposed ANSI C standard.

Lafore, Robert. *Microsoft C Programming for the IBM*. Indianapolis, Indiana: Howard W. Sams & Company, 1987.

The first half of the book teaches C. The second half concentrates on specifics of the PC environment, such as BIOS calls, memory, and video displays.

Plum, Thomas. *Learning to Program in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, 1983.

A widely used introductory college text on computer programming in C.

Schustack, Steve. *Variations in C*. Redmond, Washington: Microsoft Press, 1985.

An intermediate-level guide to developing business applications in C.

Waite, Mitchell, Stephen Prate, Bryan Costales, and Harry Henderson (The Waite Group). *Microsoft QuickC Programming*. Redmond, Washington: Microsoft Press, 1988.

Beginning- to intermediate-level C programming, with special emphasis on the QuickC Compiler.

Ward, Robert. *Debugging C*. Indianapolis, Indiana: Que Corporation, 1986.

An advanced guide to the theory and practice of debugging C programs.

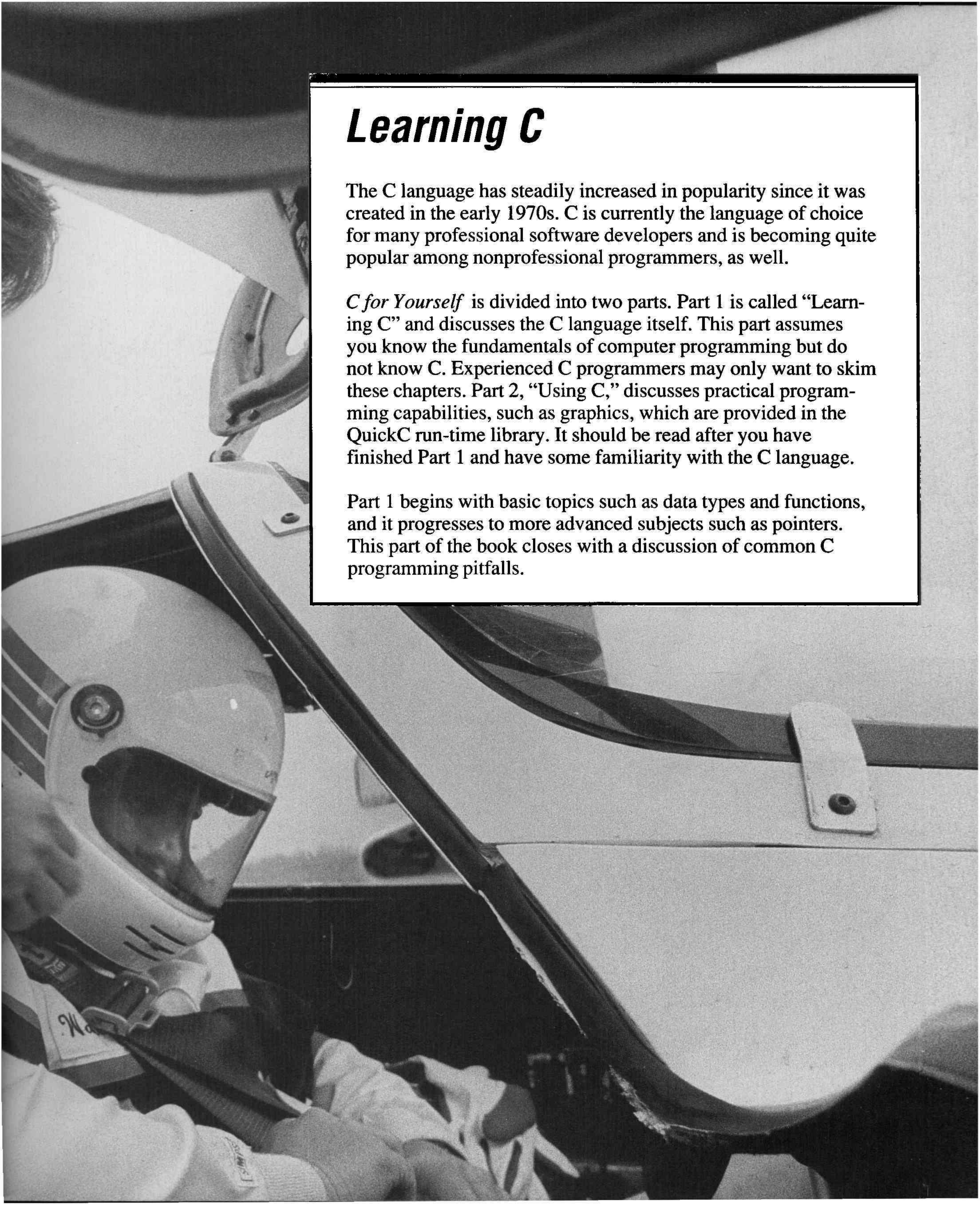
Wilton, Richard. *Programmer's Guide to PC and PS/2 Video Systems*. Redmond, Washington: Microsoft Press, 1987.

An advanced guide to all the PC and PS/2® video modes.

PART 1
Learning C

CHAPTERS


MOTOR OIL



Learning C

The C language has steadily increased in popularity since it was created in the early 1970s. C is currently the language of choice for many professional software developers and is becoming quite popular among nonprofessional programmers, as well.

C for Yourself is divided into two parts. Part 1 is called "Learning C" and discusses the C language itself. This part assumes you know the fundamentals of computer programming but do not know C. Experienced C programmers may only want to skim these chapters. Part 2, "Using C," discusses practical programming capabilities, such as graphics, which are provided in the QuickC run-time library. It should be read after you have finished Part 1 and have some familiarity with the C language.

Part 1 begins with basic topics such as data types and functions, and it progresses to more advanced subjects such as pointers. This part of the book closes with a discussion of common C programming pitfalls.

CHAPTER 1

Anatomy of a C Program

As a knowledgeable programmer, you'll probably be tempted to immerse yourself in C immediately. But before taking that plunge, you should know the basic model for all C programs. This chapter sketches the anatomy of a C program without getting bogged down in formal definitions or exceptions to the rules. (You'll find plenty of rules in the chapters that follow.)

The discussion revolves around a short, reasonably typical C program named VOLUME.C. To get comfortable with the look of C programs, as well as the basic ideas that shape them, refer to VOLUME.C frequently as you read.

A Typical C Program

VOLUME.C is a simple program that calculates the volume of a sphere and prints the following result on the screen:

```
Volume: 113.040001
```

Like all of the sample programs in this book, you'll find VOLUME.C in QuickC's online help. The "Introduction" explains how to load sample programs. Figure 1.1 illustrates the VOLUME.C program.

Compiler directives

Comment

Includes the STDIO.H file

Defines the symbolic constant PI

Function prototype

Definition of main function

Declaration of local variables

Function calls

Definition of sphere function

Declaration of local variable

Returns value to calling statement

```
/* VOLUME.C: Calculate sphere's volume. */
#include <stdio.h>
#define PI 3.14

float sphere( int rad );

main()
{
   float volume;
   int radius = 3;
   volume = sphere( radius );
   printf( "Volume: %f\n", volume );
}

float sphere( int rad )
{
   float result;
   result = rad * rad * rad;
   result = 4 * PI * result;
   result = result / 3;
   return result;
}
```

Figure 1.1 The VOLUME.C Program

Comments

The first line in VOLUME.C is a comment:

```
/* VOLUME.C: Calculate sphere's volume. */
```

Comments make a program more readable.

In C, a comment begins with a slash-asterisk (/*) and ends with an asterisk-slash (*/). Because C is a compact language with very few keywords, comments play an important role in making your programs readable.

You can't nest comments (put one comment inside another). The following line creates a syntax error:

```
/* Error! /* You can't */ nest comments in C. */
```

Statements

C statements always end with semicolons. Here is a statement from the VOLUME.C program:

```
result = 4 * PI * result;
```

Statement blocks are enclosed in braces.

You can also enclose a group of statements in braces, making a "statement block." Statement blocks contain related statements, such as the statements in the body of a function.

The C language ignores white space (tabs, blanks, and line breaks) in your source program, so you can arrange your program in almost any style. However, a few de facto rules help promote readability. A typical C program is written with one statement per line. Braces align vertically, and statements inside braces are indented. The "Introduction" describes the programming style used in this book.

Keywords and Names

C is a case-sensitive language (it distinguishes between uppercase and lowercase letters). All of C's keywords are spelled completely in lowercase; online help contains a complete list of C keywords.

You can declare names in any combination of either case, but many programmers prefer to use lowercase for variable and function names, saving uppercase for declaring symbolic constants. (A "symbolic constant" is a descriptive name that represents a constant value. In VOLUME.C, `PI` is a symbolic constant.)

Preprocessor Directives

A preprocessor directive is a command to the QuickC compiler.

Not every line in a C program is an executable statement. Programs can also contain "preprocessor directives"—commands for the QuickC compiler. A directive begins with a number sign (#) and does not end with a semicolon.

The second and third lines of VOLUME.C contain preprocessor directives. The **#include** directive in the second line tells QuickC to insert the file STDIO.H when it compiles VOLUME.C:

```
#include <stdio.h>
```

STDIO.H is one of the many "header files" supplied with QuickC. Header files contain declarations and definitions required by C library functions. ("Library functions" are supplied with QuickC rather than written by you.) In the VOLUME.C program, the **printf** library function requires information from the STDIO.H header file.

The **#define** directive in the third line defines a symbolic constant named `PI`:

```
#define PI 3.14
```

Wherever `PI` appears later in the source program, QuickC substitutes the text `3.14`. The text can be any combination of letters, digits, or other characters. The effect is much like a search and replace operation in a word processor.

Functions

A function performs a specific task and can also return a value.

Functions are the building blocks of C. Every C program has at least one function, and every executable C statement appears inside some function or another. In plain English, a "function" is a group of statements that performs a specific task and often returns a value to the statement that calls it.

C functions serve the same purposes as QuickPascal procedures and functions or BASIC **SUB** and **FUNCTION** procedures. They allow you to write well-organized programs that perform different tasks in separate parts.

The C language has no input/output statements.

C also uses functions to perform all input and output (I/O). Unlike other high-level languages, C has no I/O statements such as **PRINT** or **READ**. Instead, all I/O is done by calling C library functions such as **printf**.

Every C program has a function named main.

The VOLUME.C program contains two functions, named **main** and `sphere` (see Figure 1.1). The main execution section of every C program is itself a function named **main,** which marks where execution starts and ends. When you run VOLUME.C, execution starts at the beginning of the **main** function and stops at the end of **main**.

Calling Functions

Functions can be called (executed) from anywhere in a program, and they can receive values as well as return them. A value that you pass (send) to a function is called an "argument."

Calling a C function is a simple matter. You state the name of the function and supply in parentheses any arguments you want to pass to it. You must place a comma between arguments.

The VOLUME.C program contains two function calls, one to the **printf** library function and the other to the `sphere` function, which is defined in the program. The following statement calls the **printf** function:

```
printf( "Volume: %f\n", volume );
```

The statement passes two arguments to **printf**. The first, `"Volume: %f\n"`, supplies text and some formatting information. The second, `volume`, supplies a numeric value. See "A Few Words about **printf**," below, for more information.

In C, a function does not necessarily have to return a value. It can either return a value (like a QuickPascal function) or return nothing (like a QuickPascal procedure).

When a function returns a value, the value is often assigned to a variable. The following statement from VOLUME.C calls the `sphere` function and assigns its return value to the variable `volume`:

```
volume = sphere( radius );
```

A function uses the **return** keyword to return a value. In VOLUME.C, the last statement in the `sphere` function returns the value of the variable `result` to the statement that calls `sphere`:

```
return result;
```

Declaring and Initializing Variables

You must "declare" every variable in a C program by stating its name and type. If you refer to an undeclared variable, QuickC displays an error message when you compile the program.

The following statement from VOLUME.C declares a **float** (floating-point) type variable named `volume`:

```
float volume;
```

After declaring a variable, you should "initialize" it—give it an initial value—before using it. Uninitialized variables might have any value, so they are dangerous to use. The VOLUME.C program initializes the variable `volume` by assigning it the return value from a function call:

```
volume = sphere( radius );
```

You can also initialize a variable when it is declared, a convenient and common practice. The following statement from VOLUME.C declares the variable `radius` as an **int** (integer) variable and initializes it with the value 3:

```
int radius = 3;
```

External and Local Variables

The place where you declare a variable controls where it is visible. A variable declared outside any function is "external": you can refer to it anywhere within the program. (External variables are called "global" in some other languages.)

A variable declared inside the braces of a function is "local." You can refer to it inside the function but nowhere else. In VOLUME.C, the `result` variable is declared inside the `sphere` function:

```
float sphere( int rad )
{
   float result;
   .
   .
   .
}
```

Use external variables only when necessary.

Because it is local to the `sphere` function, the `result` variable cannot be used elsewhere in VOLUME.C. Making variables local whenever possible minimizes the risk that a variable's value will be changed accidentally in some other part of the program.

When a function receives arguments, the arguments become local variables within the function. The `sphere` function requires one argument, which it names `rad`. Within the function, `rad` is a local variable.

Function Prototypes

Function prototypes allow QuickC to check every function reference for accuracy.

A function can be declared in much the same way as a variable. Function declarations, often called "prototypes," allow QuickC to do "type checking." Given the information in the prototype, QuickC can check every subsequent use of the function to make sure you pass the right number and type of arguments and use the correct return type.

A function prototype gives the following information:

- The function's name
- The type of value the function returns
- A list of arguments the function requires

The VOLUME.C program contains one function prototype, for the `sphere` function:

```
float sphere( int rad );
```

The prototype states that the `sphere` function returns a **float** (floating-point) value and requires one **int** (integer) argument.

A Few Words about printf

The VOLUME.C program, like most examples in this book, uses the **printf** library function to display text. You won't need to know all of the details of **printf** to read the rest of this book, but the examples will be easier to follow if you know a few basic concepts.

The **printf** function works like the QuickBASIC **PRINT USING** statement or the QuickPascal **Writeln** procedure. It can display string and numeric data in various formats, and it normally prints to the screen.

You can print a simple message by passing **printf** a string (characters in double quotes) as an argument:

```
printf( "Hi, Mom!" );
```

The statement prints

```
Hi, Mom!
```

The **printf** function doesn't automatically add a newline character at the end of a line. The statements

```
printf( "Le Nozze di Figaro" );
printf( " by W. A. Mozart" );
```

print the following message on one line:

```
Le Nozze di Figaro by W. A. Mozart
```

To start a new line, use the escape sequence **\n** as follows:

```
printf( "Hi,\nMom!" );
```

The statement prints two words on separate lines:

```
Hi,
Mom!
```

The **f** in **printf** stands for formatting. To print the values of variables and other items, you supply **printf** with format codes that tell **printf** the proper format for each item. The codes are placed in the first argument, which is enclosed in double quotes.

The following statement uses the **%x** code to print the integer 553 in hexadecimal format. It passes two arguments to **printf**:

```
printf( "%x", 553 );
```

The first argument (`"%x"`) contains the format code and the second argument (`553`) contains the item to be formatted. The line displays the following:

```
229
```

The **printf** function accepts several other format codes. For instance, the VOLUME.C program uses **%f** to print a floating-point number. Some programs in later chapters use **%d** to print integers or **%ld** to print long integers.

The first argument passed to **printf** can contain any combination of characters and format codes. The other arguments contain the items that you want **printf** to format. The statement

```
printf( "%d times %d = %d\n", 2, 128, 2 * 128 );
```

prints the line:

```
2 times 128 = 256
```

The **printf** function matches the format codes to the items you want to format, in left-to-right order. In the code above, the first **%d** formats the number 2, the second formats the 128, and the third formats the expression 2 * 128 (which evaluates to the number 256).

There's much more to say about **printf** and other I/O functions, but the rest can wait until you reach Chapter 11, "Input and Output," which describes I/O in detail.

Now that you've glimpsed the big picture, we can take a closer look at some specifics of C programming, beginning with Chapter 2, "Functions."

CHAPTER 2

Functions

Chapter 1, "Anatomy of a C Program," introduced functions, the building blocks of C programs. In this chapter, you'll learn how to use functions in C programs.

We begin by discussing some function basics, including the role of the **main** function. We then show you how to call functions, pass data to them, return data from them, and declare them. The chapter concludes with a brief look at old-style function declarations, which you may encounter in some programs.

Functions and Structured Programming

As we mentioned in Chapter 1, a C function is a collection of statements, enclosed in braces ({ }), which performs a particular task. It can receive arguments (data) and also return a value.

Functions allow you to program with a "divide and conquer" strategy. Rather than try to solve a large problem all at once, you break the problem into several parts and attack each one separately. This approach, known as "structured programming," allows you to write clear, reliable programs that perform separate tasks in discrete, logically contained modules. In the C language, these modules are called functions.

Functions offer several advantages. They can

- Make programs easier to write and read. All of the statements related to a task are located in one place.
- Prevent unexpected side effects by using private (local) variables that are not visible to other parts of the program.

- Eliminate unnecessary repetition of code for frequently performed tasks.
- Simplify debugging. Once the function works reliably, you can use it with confidence in many different situations.

If you know QuickPascal or QuickBASIC, you will see many similarities in the C language. A C function serves the same basic purpose as a QuickPascal function or procedure or a QuickBASIC **FUNCTION** or **SUB** procedure. In later sections, we'll note some differences between C and these languages.

The main Function

Every C program must have one and only one main function.

Every C program must have a function named **main**, which tells where program execution begins and ends. Although **main** is not a C keyword, it has only one use: naming the **main** function. A program must have only one **main** function, and you shouldn't use the name anywhere else.

Below is the simplest possible C program:

```
main()
{
}
```

The braces ({ }) mark the **main** function's beginning and end, as they do in every function. This program doesn't contain any executable statements; it simply begins and ends.

Most functions have executable statements, of course, and these appear within the function's braces. The following program contains a statement which prints `Hello, world!` on the screen:

```
main()
{
   printf( "Hello, world!\n" );
}
```

The **main** function is called by the operating system when it runs your program. While it's possible to call the **main** function in a program, you should never do so, just as you wouldn't write a QuickBASIC program containing the line

```
10 GOSUB 10
```

A program that calls **main** will start again and again in an endless loop that eventually triggers a run-time error.

Like all functions, **main** can accept arguments and return a value. Through this mechanism, your program can receive command-line arguments from DOS when it begins execution and return a value to DOS when it ends. Chapter 9, "Advanced Pointers," describes how to receive command-line arguments via **main**.

Placement and Visibility of Functions

A function is normally visible everywhere in the program.

Every C function is normally "visible" to all other functions in the same program. That is, it can call and be called by any other function. C functions can even call themselves, a process known as "recursion."

In the program below, the functions `whiz` and `bang` are visible to **main** and to each other. The **main** function can call both `whiz` and `bang`. In addition, `whiz` can call `bang`, and vice versa.

```
main()
{
}

whiz()
{
}

bang()
{
}
```

Functions can appear in any order and at almost any place in your program. Since **main** starts and ends the program's execution, this function often begins the program. But this is a readability convention, not a language requirement.

C functions can't be nested.

One place where you can't put a function is inside another function. The C language doesn't allow you to nest functions. Here C differs from QuickPascal, in which one procedure can contain other "hidden" functions or procedures. The following program causes a syntax error because the `bang` function appears within the `whiz` function:

```
main()
{
}

whiz()
{
   /* Error! Incorrect function placement */
   bang()
   {
   }
}
```

Function Definitions and Prototypes

Now that you understand some function basics, we can look at functions in more detail. A function, or more precisely, "function definition," contains several

parts. Figure 2.1 shows the parts of the `sphere` function definition from the VOLUME.C example in Chapter 1, "Anatomy of a C Program."

Return type *Name* *Argument list*

Function header

Function body

```
float sphere( int rad )
{
   float result;

   result = rad * rad * rad;
   result = 4 * PI * result;
   result = result / 3;
   return result;
}
```

Figure 2.1 Typical C Function

The function "header" specifies the type of value a function returns and the function's name. The header also contains an argument list, which specifies the arguments the function requires. The rest of the function definition—everything inside the braces—is the function "body."

The ANSI C standard, which QuickC follows, recommends that you supply a function "prototype" (declaration) for every function definition in your program. The prototype is identical to the function header except that it ends with a semicolon. Figure 2.2 shows the `sphere` function prototype from VOLUME.C.

Return type *Name* *Argument list*

```
float sphere( int rad );
```

Figure 2.2 The Sphere Function from VOLUME.C

The function prototype normally appears near the beginning of the program and serves a purpose similar to a variable declaration. It provides advance information about the function, which QuickC can use to check the accuracy of subsequent calls to the function. We'll examine prototypes in detail in "Function Prototypes," below.

Calling a Function

You call (execute) a function by stating its name. In the simplest case—when a function doesn't receive or return any data—the function call consists of the function's name, followed by an empty pair of parentheses and a semicolon. The BEEPER.C program, shown below, demonstrates this kind of function call.

```
/* BEEPER.C: Demonstrate simple function. */
#include <stdio.h>

void beep( void );

main()
{
   printf( "Time to beep\n" );
   beep();
   printf( "All done\n" );
}

void beep( void )
{
   printf( "Beep!\a\n" );
}
```

When you run BEEPER.C, the program prints:

```
Time to beep
Beep!
All done
```

As you may recall from Chapter 1, "Anatomy of a C Program," the **\n** sequence represents the newline character. The **\a** sequence is the "alert" character (ASCII 7) which makes an audible beep.

In the **main** function of BEEPER.C, the statement

```
beep();
```

calls the `beep` function. Since `beep` takes no arguments, the parentheses of the function call are empty.

The prototype and definition for the `beep` function use the **void** keyword twice, first to indicate that the function returns no value, and second to indicate that it receives no arguments. We'll return to these points later in this chapter.

A function call transfers control to that function. The statements within the function's braces execute in order until the function ends. Then execution resumes where it left off.

A function can end in one of two ways. The `beep` function above ends by "falling off" the closing brace of the function definition. A function can also end by executing a **return** statement, which we discuss later in the section "Returning Values from Functions."

Figure 2.3 illustrates the flow of control in BEEPER.C.

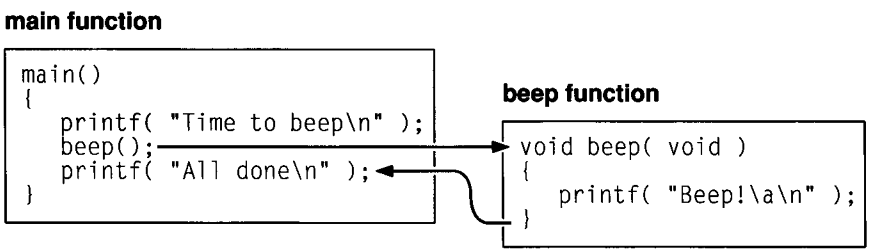


Figure 2.3 Calling a C Function

Passing Arguments to a Function

If a function requires arguments, you list them in the parentheses of the function call. In the BEEPER1.C program below, we revise the `beep` function from BEEPER.C to take one argument.

```
/* BEEPER1.C: Demonstrate passing arguments. */
#include <stdio.h>

void beep( int num_beep );

main()
{
   printf( "Time to beep\n" );
   beep( 5 );
   printf( "All done\n" );
}

void beep( int num_beep )
{
   while( num_beep > 0  )
   {
      printf( "Beep!\a\n" );
      num_beep = num_beep - 1;
   }
}
```

The function definition states what kind of arguments the function expects. In the `beep` function definition, the header,

```
void beep( int num_beep )
```

states that `beep` expects one **int** (integer) argument named `num_beep` (number of beeps).

The statement that calls `beep`,

```
beep( 5 );
```

gives the value 5 in parentheses, passing that value as an argument. Figure 2.4 shows argument passing in BEEPER1.C.

main function

```
main()
{
   printf( "Time to beep\n" );
   beep( 5 );
   printf( "All done\n" );
}
```

5

beep function

```
while( num_beep > 0 )
{
   printf( "Beep!\a\n" );
   num_beep = num_beep - 1;
}
```

Figure 2.4 Passing an Argument to a Function

Function arguments are assigned to local variables inside the function.

When `beep` receives the value 5, the function automatically assigns the value to `num_beep`, which the function can then treat as a local variable. In this case, the function uses `num_beep` as a loop counter to repeat the statement

```
printf( "Beep!\a\n" );
```

`num_beep` times. (The C **while** loop is very similar to **WHILE** loops in QuickBASIC or QuickPascal. You don't need to know the details of loops for now; they're explained in Chapter 3, "Flow Control.")

If a function expects more than one argument, you separate the arguments with commas. For instance, the statement

```
printf( "%d times %d equals %d\n", 2, 16, 2 * 16 );
```

passes four arguments to the **printf** function. The first argument is the string

```
"%d times %d equals %d\n"
```

The second and third arguments are constants (`2` and `16`). The fourth argument is an expression (`2 * 16`) that evaluates to a constant.

Arguments Versus Parameters

In the C language, a value passed to a function is called either an "argument" or a "parameter," depending on viewpoint. From the viewpoint of the statement that calls the function, the value is an argument. In the view of the function receiving it, the value is a parameter.

Thus, in BEEPER1.C, the following function call passes an argument to the `beep` function:

```
beep( 5 );
```

Looking at the same value from the receiving end, the header of the `beep` function declares a parameter named `num_beep` as follows:

```
void beep( int num_beep );
```

The argument and parameter refer to the same value—in this case, the value 5. The naming distinction is just a matter of viewpoint, similar to the way you call a letter outgoing mail if you're sending it, or incoming mail if you're receiving it.

Assigning Parameters

When you list a parameter in the function header, it becomes a local variable within the function. This process is easy to follow when it involves only one argument, as in the BEEPER1.C program above. The function call passes one value, which the function assigns to one variable. The variable can be treated like any other variable declared within the function.

There is a one-to-one correspondence between arguments and parameters.

If a function takes more than one argument, the values are passed in order. The first argument in the function call is assigned to the first variable. the second argument is assigned to the second variable, and so on.

The SHOWME.C program below demonstrates this process. Its `showme` function takes three arguments. The **main** function defines three integer variables and passes their values to `showme`, which prints the values that it receives. (You normally wouldn't write a function just to print one line, of course. We'll add more to SHOWME.C in a later revision.)

```
/* SHOWME.C: Demonstrate passing by value. */
#include <stdio.h>

void showme( int a, int b, int c );
```

```
main()
{
   int x = 10, y = 20, z = 30;
   showme( z, y, x );
}

void showme( int a, int b, int c )
{
   printf( "a=%d b=%d c=%d", a, b, c );
}
```

Here's the output from SHOWME.C:

```
a=30 b=20 c=10
```

The function call in SHOWME.C passes the values of `z`, `y`, and `x` in the order listed:

```
showme( z, y, x );
```

Functions receive parameters in the order they are passed.

These values are assigned, in the same order, to the parameters listed in the `showme` function header:

```
void showme( int a, int b, int c )
```

The position of the parameters, not their names, controls which arguments the parameters receive. The first argument (`z`) listed in the function call is assigned to the first parameter (`a`) in the function header, the second argument (`y`) to the second parameter (`b`), and so on. Figure 2.5 shows this process.

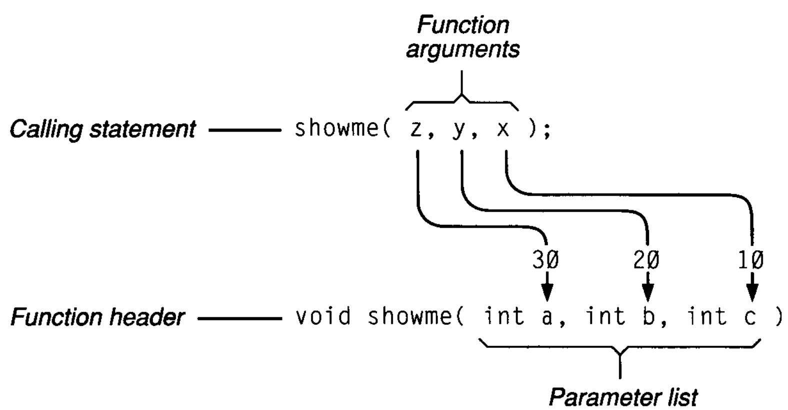


Figure 2.5 Assigning Parameters in SHOWME.C

Passing by Value

The C language passes copies of function arguments.

In C, all function arguments (except arrays) are passed "by value" rather than "by reference." That is, a function receives a local copy of each argument, not the argument itself. These copies are local variables within the function. They are created and initialized automatically when the function begins, and they disappear when it ends. Like all local variables, their values can be changed without affecting variables elsewhere in the program.

We can clarify this point by adding a few statements to the SHOWME.C program. The new program, SHOWMORE.C, will change the values of the local variables in the `showme` function without changing the values of the original variables.

```
/* SHOWMORE.C: Demonstrate passing by value. */
#include <stdio.h>

void showme( int any, int old, int name );

main()
{
   int x = 10, y = 20, z = 30;
   showme( z, y, x );
   printf( "  z=%d   y=%d    x=%d\n", z, y, x );
}

void showme( int any, int old, int name )
{
   printf( "any=%d old=%d name=%d\n", any, old, name );
   any = 55;
   old = 66;
   name = 77;
   printf( "any=%d old=%d name=%d\n", any, old, name );
}
```

Here is the output from SHOWMORE.C:

```
any=30 old=20 name=10
any=55 old=66 name=77
  z=30   y=20    x=10
```

First, note that the `showme` function in SHOWMORE.C uses new names (`any`, `old`, and `name`) when assigning the parameters it receives:

```
void showme( int any, int old, int name )
```

Function parameters can have any legal variable names.

Because these variables are local to the function, they can have any legal names. (The rules for variable names are described in Chapter 4, "Data Types.") The `showme` function prints the values of its parameters immediately after assigning them:

```
printf( "any=%d old=%d name=%d", any, old, name );
```

Then the function assigns new values to the variables and prints them again:

```
any = 55;
old = 66;
name = 77;
printf( "any=%d old=%d name=%d", any, old, name );
```

Local variables are private to the function containing them.

Changing the local variables in the `showme` function doesn't affect the original variables in the **main** function. Remember, a variable defined inside a function is only visible inside that function. After control returns to **main**, SHOWMORE.C prints the values of the original variables:

```
printf( "  z=%d   y=%d    x=%d\n", z, y, x );
```

As the program output shows, the original values are unchanged:

```
z=30   y=20    x=10
```

We'll say more about the visibility of variables in Chapter 5, "Advanced Data Types." For now, just remember that when you pass a value to a function, the function makes a local copy of that value. The local copy can be manipulated without changing the original.

NOTE *In QuickPascal, you can pass either the value of an argument or the argument's address. In C, function arguments are only passed by value. However, that value can be an address. Chapter 8, "Pointers," explains how to pass addresses to functions.*

Returning Values from Functions

The return keyword ends a function and can return one value.

Most C functions return a value. This is done with the **return** statement, which also ends the function. The VOLUME.C program from Chapter 1, "Anatomy of a C Program," (see Figure 2.1) contains such a statement. In that program, the `sphere` function returns the value of the variable `result` as follows:

```
return result;
```

The following statement in the **main** function of VOLUME.C calls the `sphere` function and assigns its return value to the variable `volume`:

```
volume = sphere( radius );
```

Figure 2.6 shows the flow of control as the `sphere` function returns a value in VOLUME.C.

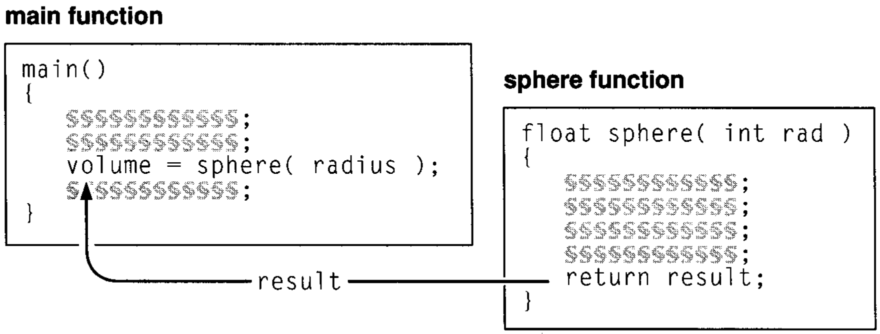


Figure 2.6 Returning a Value from a Function

A **return** statement can only return a single value. If a function needs to return multiple values, the normal method is to use pointers—a major topic that we'll discuss in Chapter 8, "Pointers."

NOTE *In QuickPascal, a function returns a value and a procedure does not. The same distinction applies to QuickBASIC **FUNCTION** and **SUB** procedures, respectively. In the C language, a function can do both. It can return a value or return nothing.*

A function can contain more than one **return** statement, as shown below:

```
if( error == 0 )
   return 0;
else
   return 1;
```

The code returns a different value in different cases. It returns the value 0 when the variable `error` is 0 and the value 1 when `error` is nonzero. (In C, the **if** and **else** statements work much like those in other languages. Chapter 3, "Flow Control," explains these statements.)

A return statement can appear anywhere and need not return a value.

You can place the **return** keyword anywhere within a function, and the statement need not necessarily return a value. In the following fragment, the naked **return** statement simply ends the function if the value of `count` exceeds 500:

```
if( count > 500 )
   return;
else
   /* execute more statements... */
```

A **return** statement ends the function immediately, no matter where it appears. In the function shown below, the statements following the **return** never execute:

```
void do_nothing( void )
{
   return;
   /* The following statements do not execute */
   printf( "This function " );
   printf( "prints nothing.\n" );
}
```

If a function doesn't return a value, and you want the function to end by falling off its closing brace, no **return** statement is needed. This method is used to end the `beep` function in BEEPER.C, discussed earlier in this chapter:

```
void beep( void )
{
   printf( "Beep!\a\n" );
}
```

You could add a **return** to the end of this function, but it's not necessary. The function ends automatically.

Using Return Values

Function return values are often assigned to variables.

A function's return value can be used in the same way you would use any value of its type. In the VOLUME.C program from Chapter 1, "Anatomy of a C Program," the statement that calls `sphere` assigns the function's return value to the variable `volume`:

```
volume = sphere( radius );
```

If there's no need to save the return value, you can use it directly. You may have noticed that the variable `volume` isn't really needed in the VOLUME.C program, which simply prints the variable's value and ends. Most programmers would make the program more compact by replacing the two statements

```
volume = sphere( radius );
printf( "Volume: %f\n", volume );
```

with this one:

```
printf( "Volume: %f\n", sphere( radius ) );
```

The second version puts the `sphere` function call right in the **printf** statement, eliminating the superfluous variable. Instead of assigning the return value to a variable and passing that variable's value to **printf**, the statement uses the value directly. (The `sphere` function is called first. Then the return value from `sphere` is passed as an argument to the **printf** function.)

While this change streamlines the program, it also makes the code a little harder to follow. If you don't read carefully, you might overlook the fact that the **printf** function call contains another function call.

Unused return values are discarded.

Occasionally, you may have no use for a function's return value. The **printf** function, for example, returns the number of characters it displayed, but few programs need this information. If you don't use a return value, it's discarded.

You should never ignore the error codes that library functions return to show whether the function succeeded. See Chapter 10, "Programming Pitfalls," for more information about library function return values.

Declaring a Function's Return Type

Thus far, we have explained how a function can return a value—and how the calling statement can use that value—without paying much attention to what type of value the function returns. (The C language supports various data types, such as **int** for integer values, and **float** for floating-point values. Chapter 4 describes data types in detail.)

The return type is important because it controls what the function returns. If a function returns an integer when you expect a floating-point value, your program may not work correctly.

A function's prototype and definition control what type of value it returns.

The function's return type is specified in its prototype and definition. Below are the prototype and definition of the `sphere` function from the VOLUME.C program in Chapter 1, "Anatomy of a C Program." They specify that the function returns a **float** value.

```
float sphere( int rad );  /* function prototype */

float sphere( int rad )   /* function header */
```

The type name (here, **float**) in front of the function name shows what type of value the function returns. If the `sphere` function returned an **int** value, its prototype and header would look like this:

```
int sphere( int rad ); /* function prototype */

int sphere( int rad )  /* function header */
```

Use the void type name to show a function returns no value.

You should declare the return type for every function—even for functions that don't return a value. These functions are declared with the **void** type name. In the SHOWME.C program, shown above, the prototype of the `showme` function follows this pattern:

```
void showme( int a, int b, int c );
```

The **void** that precedes the function name indicates that `showme` returns nothing.

Function Prototypes

Function prototyping is the major innovation of the ANSI standard for C. As we mentioned earlier, a function prototype gives the same information as the function's header: the name of the function, the type of value the function returns, and the number and type of parameters the function requires.

Function prototypes allow QuickC to check function references for accuracy.

Function prototypes normally appear near the start of the program, before the first function definition. Given the information in the prototype, QuickC can perform "type checking." It checks each reference to the function—its definition, as well as every function call—to make sure that the reference uses the right number and type of arguments and the correct return value. Without type checking, it's easy to create bugs by passing the wrong type of value to a function or assuming the wrong return type.

C programs normally include one prototype for each function they define, except the **main** function. Most programmers don't bother to prototype **main** unless the program receives command-line arguments or returns a value to DOS when it ends. (Command-line arguments are discussed in Chapter 9, "Advanced Pointers.")

Here is the function prototype for the `sphere` function in VOLUME.C:

```
float sphere( int rad );
```

You can see that `sphere` expects a single **int**-type parameter and returns a value of type **float**. On the other hand, the prototype for `showme` in SHOWME.C indicates that `showme` expects three **int**-type parameters and returns nothing:

```
void showme( int a, int b, int c );
```

It's common to use the same parameter names in both the function prototype and the function header. In SHOWME.C, for instance, the `showme` function prototype,

```
void showme( int a, int b, int c );
```

uses the names `a`, `b`, and `c`, as does the header for that function,

```
void showme( int a, int b, int c )
```

Using the same names in both parameter lists makes the program more readable, but it's not a language requirement. The names in the prototype's parameter list are merely cosmetic. You can use any names you choose, or even omit the names completely. The prototype in SHOWME.C works just as well when written

```
void showme( int, int, int );
```

as when you supply the names `a`, `b`, and `c`. Both versions fully specify the number (three) and type (**int**) of the parameters the function expects.

Prototyping Functions without Parameters

If a function doesn't expect any parameters, you might be tempted to leave its parameter list blank. But it's better to put **void** in its parameter list, as shown here:

```
void beep( void );
```

The **void** in parentheses specifies that the `beep` function requires no parameters. If you leave the parentheses empty, the compiler draws no conclusion about what parameters the function takes and won't be able to detect an error if you mistakenly pass an argument to the function.

Prototyping Functions with Variable Parameters

Some functions, such as the **printf** library function, can handle a variable number of parameters. This capability can make functions more flexible. As earlier examples have shown, the **printf** function can take one parameter or several, depending on how many values you need to print.

To declare a function with a variable number of parameters, end the parameter list with a comma and an ellipsis (`, . . .`). The following prototype, for example, declares that the `sample` function expects at least one **int**-type parameter and zero or more additional parameters:

```
void sample( int a,... );
```

The ellipsis stands for an unspecified number of parameters with types that are also unspecified. The parameter list in the function header should follow the same pattern.

Don't declare a variable number of parameters unless it's necessary. Giving this sort of prototype for a function that takes a fixed number of parameters defeats the prototype's main purpose. QuickC can't perform type checking for parameters you leave out of a prototype.

Old-Style Function Declarations and Definitions

This book explains how to declare and define functions under the ANSI standard for C, which is now the norm. The original C language used slightly different rules for function declarations and definitions. QuickC can compile these "old-style" programs, but the ANSI standard recommends you use the full function prototypes we just described.

Still, you may encounter old-style function declarations and definitions in many existing C programs. So, you should be familiar with the style.

We'll use the VOLUME.C program from Chapter 1, "Anatomy of a C Program," to demonstrate the old style. First, here's the ANSI-style program presented in Chapter 1:

```
/* VOLUME.C: Calculate sphere's volume. */
#include <stdio.h>
#define PI 3.14

float sphere( int rad );

main()
{
   float volume;
   int radius = 3;
   volume = sphere( radius );
   printf( "Volume: %f\n", volume );
}

float sphere( int rad )
{
   float result;
   result = rad * rad * rad;
   result = 4 * PI * result;
   result = result / 3;
   return result;
}
```

The same program written in the old style would look something like this:

```
/* OLDSTYLE.C: Old-style function. */

#include <stdio.h>
#define PI 3.14

float sphere();

main()
{
   float volume;
   int radius = 3;
   volume = sphere( radius );
   printf( "Volume: %f\n", volume );
}

float sphere( rad )
int rad;
{
   float result;
   result = rad * rad * rad;
   result = 4 * PI * result;
   result = result / 3;
   return result;
}
```

You'll notice two distinct differences. First, the old style doesn't allow a parameter list in the function declaration. In the ANSI version, VOLUME.C, the declaration of the `sphere` function specifies that the function takes a single **int** parameter:

```
float sphere( int rad );
```

The corresponding declaration in OLDSTYLE.C omits the parameter list:

```
float sphere();
```

An old-style function declaration cannot provide any information about the function's parameters.

The other change is in the way the function definition lists parameters. In VOLUME.C, the function header lists the same information as the function prototype, giving the type (**int**) and name (`rad`) of the function's parameter:

```
float sphere( int rad )
{
    .
    .
    .
}
```

In OLDSTYLE.C, the function header gives the parameter's name (`rad`), but not its type. The parameter's type is declared in a statement directly below the function header (and before the left brace that begins the function body):

```
float sphere( rad )
int rad;
{
    .
    .
    .
}
```

The rest of OLDSTYLE.C is identical to VOLUME.C.

Now that you understand the basics of functions, we can turn our attention to the C statements that a function can contain, beginning with flow-control statements, the subject of the next chapter.

CHAPTER 3

Flow Control

Flow control—diverting execution by looping and branching—is one area where C closely resembles other languages. If you know how to loop and branch in QuickBASIC or QuickPascal, learning the C equivalents is mainly a matter of adjusting to somewhat different syntax. Here, as elsewhere, C never uses two keywords when one will do. For instance, C has no "then" keyword. Instead, it uses simple punctuation.

This chapter has two parts. The first part examines the looping statements: **while**, **do**, and **for**. The second part describes the decision-making statements: **if**, **else**, **switch**, **break**, **continue**, and **goto**.

Loops: while, do, and for

This section discusses the C statements that create loops: **while**, **do**, and **for**. These loops repeat while a condition is true or for a set number of times. We'll begin with the simplest loop, the **while** statement.

The while Statement

A while loop evaluates its test expression before executing the body of the loop.

A **while** loop repeats as long as a given condition remains true. It consists of the **while** keyword followed by a test expression in parentheses and a loop body, as shown in Figure 3.1. The "test expression" can be any C expression and is evaluated before the loop body is executed. The loop body is a single statement or a statement block that executes once every time the loop is iterated. The distinguishing feature of a **while** loop is that it evaluates the test expression before it executes the loop body, unlike the **do** loop, which we'll examine next.

Test expression

Loop body

```
while( test > 0 )
{
   printf( "test = %d\n", test );
   test = test - 2;
}
```

Figure 3.1 Elements of a while Loop

We've incorporated the simple **while** loop from Figure 3.1 in the WHILE.C program, shown below.

```
/* WHILE.C: Demonstrate while loop. */

#include <stdio.h>

main()
{
   int test = 10;

   while( test > 0 )
   {
      printf( "test = %d\n", test );
      test = test - 2;
   }
}
```

Here is the output from WHILE.C:

```
test = 10
test = 8
test = 6
test = 4
test = 2
```

In WHILE.C, if the variable `test` is positive when the loop begins, the test expression evaluates as true and the loop executes. If `test` has a 0 or negative value when the loop starts, the test expression is false; the loop body does not execute and the action falls through to the statement that follows the loop.

(Chapter 6, "Operators," explains true and false values. For now, it's enough to know that an expression is evaluated as false if it equals 0. Any nonzero value is true.)

The loop body in WHILE.C happens to be a statement block enclosed in braces. If the loop body is a single statement, as in the following code, no braces are needed.

```
main()
{
   int test = 10;
   while( test > 0 )
      test = test - 2;
}
```

Occasionally, you'll see a **while** loop with a test expression such as

```
while( 1 )
```

or

```
#define TRUE 1
    .
    .
    .
while( TRUE )
```

The test expressions above are always true, creating an indefinite loop that never ends naturally. You can only terminate this kind of loop with some overt action, usually by executing a **break** statement. (See "The break Statement" later in this chapter.) You can use such a loop to repeat an action for an indefinite time period—until a certain key is pressed, for instance.

The do Statement

A **do** loop is simply a **while** loop turned on its head. First comes the loop body, then the test expression. Unlike a **while** loop, a **do** loop always executes its loop body at least once.

A do loop always executes at least once.

The difference is important. A **while** statement evaluates the test expression before it executes the loop body. If the test expression in a **while** statement is false, the loop body doesn't execute at all. A **do** statement, on the other hand, evaluates the test expression after executing the loop body. Thus, the body of a **do** statement always executes at least once, even if the test expression is false when the loop begins.

Figure 3.2 contrasts the **while** loop from WHILE.C with a comparable **do** loop to emphasize this difference. You'll notice that the **do** keyword comes right before the loop body, which is followed by the **while** keyword and a test expression—the same test expression that WHILE.C uses. Notice the semicolon that ends the **do** loop. A **do** loop always ends with a semicolon; a **while** loop never does.

Test expression

```
            while( test > 0 )
            {
Loop body      printf( "test = %d\n", test );
               test = test - 2;
            }

            do
            {
Loop body      printf( "test = %d\n", test );
               test = test - 2;
            } while( test > 0 );
```

Test expression

Figure 3.2 Comparison of the do and the while Loops

The DO.C program below uses the **do** loop from Figure 3.2 to perform the same action that WHILE.C does.

```
/* DO.C: Demonstrate do loop. */

#include <stdio.h>

main()
{
   int test = 10;
   do
   {
      printf( "test = %d\n", test );
      test = test - 2;
   } while( test > 0 );
}
```

DO.C gives the same output as WHILE.C:

```
test = 10
test = 8
test = 6
test = 4
test = 2
```

The programs do not give the same output if you modify them so that the value of `test` is 0 when the loop starts. In that case, the loop body in DO.C executes once, but the loop body in WHILE.C doesn't execute at all. You should only use a **do** loop when you always want the loop body to execute at least once.

The for Statement

As in QuickBASIC or QuickPascal, the **for** statement in C is often used to repeat a statement a set number of times. Let's begin with a simple example. The FORLOOP.C program, shown below, uses **for** to repeat a **printf** statement five times.

```
/* FORLOOP.C: Demonstrate for loop. */

#include <stdio.h>

main()
{
   int test;
   for( test = 10; test > 0; test = test - 2 )
      printf( "test = %d\n", test );
}
```

FORLOOP.C gives the same output as WHILE.C and DO.C:

```
test = 10
test = 8
test = 6
test = 4
test = 2
```

Figure 3.3 shows the parts of the **for** loop in FORLOOP.C.

Initializing expression   *Test expression*   *Modifying expression*

```
for( test = 10; test > 0; test = test - 2 )
   printf( "test = %d\n", test );
```

Loop body

Figure 3.3 Elements of the for Loop

A **for** statement is more complex than a **while** or **do** statement. The part in parentheses can contain three expressions separated by semicolons:

- An "initializing expression" that often initializes a loop counter
- A "test expression" that states how long the loop continues
- A "modifying expression" that often modifies a loop counter

Like the test expression in a **while** statement, the test expression in a **for** statement causes the loop to continue as long as the test expression evaluates as true.

All of the expressions in the parentheses of a **for** statement are optional. If you omit the test expression (the second one), the statement repeats indefinitely. In the following program, for instance, all of the expressions in the parentheses of the **for** loop are empty:

```
main()
{
   for( ; ; )
      printf( "Hi, Mom!\n" );
}
```

The loop above repeats indefinitely because it has no test expression that specifies when to end the loop. It has the same effect as the following **while** loop, whose test expression is always true:

```
main()
{
   while( 1 )
      printf( "Hi, Mom!\n" );
}
```

You can use multiple expressions for either the initializing expression or the modifying expression, as in FORLOOP1.C:

```
/* FORLOOP1.C: Demonstrate multiple expressions. */

#include <stdio.h>

main()
{
   int a, b;
   for( a = 256, b = 1; b < 512; a = a / 2, b = b * 2 )
      printf( "a = %d  b = %d\n", a, b );
}
```

The output from FORLOOP1.C appears below:

```
a = 256  b = 1
a = 128  b = 2
a = 64   b = 4
a = 32   b = 8
a = 16   b = 16
a = 8    b = 32
a = 4    b = 64
a = 2    b = 128
a = 1    b = 256
```

In the FORLOOP1.C program, the initializing expression of the **for** loop initializes two variables (a and b) instead of one. The modifying expression changes the values of the same two variables. Use commas to separate multiple expressions in cases such as this.

Although **for** and **while** might seem quite different, they're interchangeable in most cases. The FORLOOP2.C program demonstrates this principle. Both loops, while constructed differently, produce the same effect—printing the numbers from 0 through 9.

```
/* FORLOOP2.C: Demonstrate similarity of for and while. */

#include <stdio.h>

main()
{
   int count;

   for( count = 0; count < 10; count = count + 1 )
      printf( "count = %d\n", count );

   count = 0;
   while( count < 10 )
   {
      printf( "count = %d\n", count );
      count = count +1;
   }

}
```

The two loops in FORLOOP2.C function identically. The **for** loop prints the numbers from 0 through 9:

```
for( count = 0; count < 10; count = count + 1; )
   printf( "count = %d\n", count );
```

as does the **while** loop:

```
count = 0;
while( count < 10 )
{
   printf( "count = %d\n", count );
   count = count + 1;
}
```

Most programmers prefer **for** over **while** in a case like this, because **for** groups all the loop-control statements in one place. The **for** statement is also appropriate when you need to initialize one or more values at the beginning of the loop. The **while** and **do** statements are more appropriate for cases in which the value used in the test expression has already been initialized.

Decision-Making Statements: if, else, switch, break, continue, and goto

The C language provides six statements for decision making: **if, else, switch, break, continue**, and **goto**. Like their counterparts in other languages, these statements transfer control based on the outcome of a logical test.

The if Statement

The body of an if statement executes when its test expression is true.

An **if** statement consists of the **if** keyword followed by a test expression in parentheses and a second statement. The second statement is executed if the test expression is true, or skipped if the expression is false.

The IFF.C program contains a simple **if** test. It prints a prompt and waits for you to press a key.

```
/* IFF.C: Demonstrate if statement. */

#include <stdio.h>
#include <conio.h>

main()
{
   char ch;
   printf( "Press the b key to hear a bell.\n" );
   ch = getch();
   if( ch == 'b' )
      printf( "Beep!\a\n" );
}
```

In the IFF.C program, the statement

```
ch = getch();
```

calls the **getch** library function to get a keypress from the keyboard and then assigns the result to the variable `ch`. If you press the b key, the program prints

```
Beep!
```

and sounds a beep. (To simplify the code, IFF.C tests only for a lowercase b character. A program would normally use a library function such as **tolower** to test for both upper and lowercase.)

Figure 3.4 illustrates the parts of the **if** statement in the IFF.C program.

Test expression

```
if( ch == 'b' )
   printf( "Beep!\a\n" );
```

Statement body

Figure 3.4 Elements of an if Statement

The test expression of the **if** statement

```
ch == 'b'
```

The equality operator (==) tests if values are equal.

is true when the variable `ch` equals the letter b. In C the equality operator (==) tests if two values are equal. (Chapter 6 discusses operators.)

The body of the **if** statement in IFF.C happens to be a single statement, but the body can also be a statement block, as in the following fragment:

```
if( ch == 'b' )
{
   printf( "Beep!\a\n" );
   printf( "You pressed the 'b' key\n" );
}
```

You can also nest **if** statements, as shown below:

```
if( ch == 'b' )
{
   printf( "Beep!\a\n" );
   beep_count = beepcount + 1;
   if( beep_count > 10 )
   {
      printf( "More than 10 beeps...\n" );
      if( beep_count > 100 )
         printf( "Don't wear out the 'b' key!\n" );
   }
}
```

The code nests three **if** statements. The first **if** tests whether `ch` equals the letter b; the second tests whether the variable `beep_count` is greater than 10. The third tests whether `beep_count` exceeds 100.

The else Clause

An else clause can follow an if statement.

The **else** keyword is used with **if** to form an either–or construct that executes one statement when an expression is true and another when it's false. The ELSE.C program demonstrates **else** by adding an **else** clause to the code from IFF.C. It sounds a beep and prints `Beep!` if you type the letter b, or it prints `Bye bye` if you type any other letter.

```
/* ELSE.C: Demonstrate else clause. */

#include <stdio.h>
#include <conio.h>

main()
{
   char ch;
   printf( "Press the b key to hear a bell.\n" );
   ch = getch();
   if( ch == 'b' )
      printf( "Beep!\a\n" );
   else
      printf( "Bye bye\n" );
}
```

To create an else–if construct, place an if statement after an else.

The C language has no "elseif" keyword, but it's easy to create the same effect, because the statement that follows **else** can be any C statement—including another **if** statement. The ELSE1.C program uses **if** and **else** to test three conditions. It sounds a beep when you type the letter b, it prints `Enter` when you press the ENTER key, or it prints `Bye bye` when you press any other key.

```
/* ELSE1.C: Demonstrate else-if construct. */

#include <stdio.h>
#include <conio.h>

main()
{
   char ch;
   printf( "Press the b key to hear a bell.\n" );
   ch = getch();
   if( ch == 'b' )
      printf( "Beep!\a\n" );
   else
      if( ch == '\r' )
         printf( "Enter\n" );
   else
      printf( "Bye bye\n" );
}
```

The **else** keyword is tied to the closest preceding **if** that's not already matched by an **else**. Keep this rule in mind when creating nested **if-else** constructs. (See the section "Mismatching if and else Statements" in Chapter 10, "Programming Pitfalls.")

The switch Statement

The switch statement can perform multiple branches.

The **switch** statement offers an elegant option in situations that require multiple branches. It tests a single expression that can have several values, providing a different action for each value.

One disadvantage of **if** and **else** is that they only allow one branch per keyword. The program either executes the statement that follows the **if** or **else**, or it doesn't. To perform more complex tests, you have to pile on more **if** and **else** statements, as in the ELSE1.C program above.

A program that handles keyboard input, for instance, may require several different responses to various keypresses. The ELSE1.C program above used combinations of **if** and **else** to process keyboard input. We've used a single **switch** statement in the SWITCH.C program below to do the same job:

```
/* SWITCH.C: Demonstrate switch statement. */

#include <stdio.h>
#include <conio.h>

main()
{
   char ch;
   printf( "Press the b key to hear a bell.\n" );
   ch = getch();
   switch( ch )
   {
      case 'b':
         printf( "Beep!\a\n" );
         break;
      case '\r':
         printf( "Enter\n" );
         break;
      default:
         printf( "Bye bye" );
         break;
   }
}
```

The SWITCH.C program produces the same output as ELSE1.C. Figure 3.5 illustrates the **switch** statement in SWITCH.C, comparing it with the **if-else** construct in ELSE1.C.

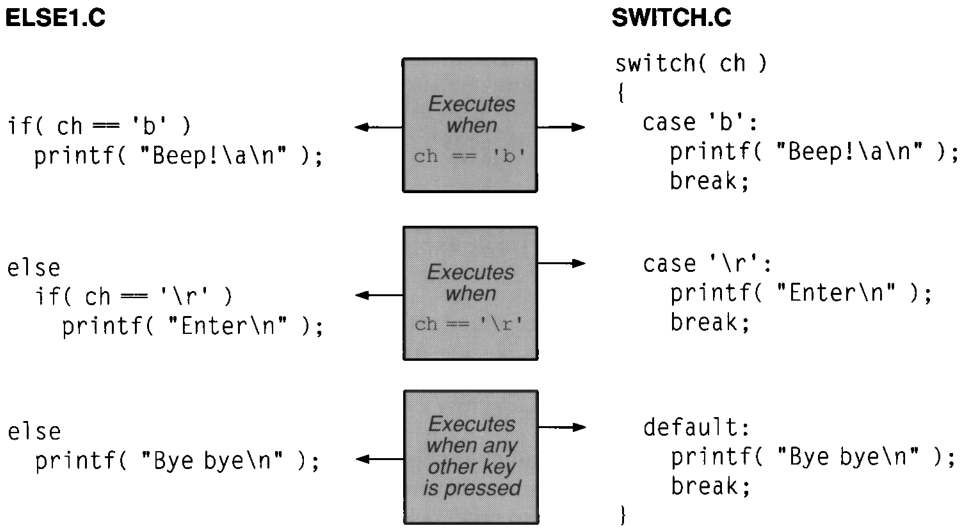


Figure 3.5 Comparison of if-else and switch Statements from ELSE1.C and SWITCH.C

As in other decision-making statements, the parentheses after the keyword contain the expression to test. This can be any expression that yields a constant value. The test expression in SWITCH.C evaluates the value of the variable `ch`:

```
switch( ch )
```

The switch statement branches to one of several labeled alternatives.

The test expression is followed by a statement block enclosed in curly braces. The block contains alternate sections of code that you want to execute under various circumstances. The program's action branches to one of the alternatives, depending on the value of the test expression.

Each alternative in the statement block starts with a "case label," which consists of the **case** keyword, a constant or constant expression, and a colon. (The only other C statement that uses labels is **goto**, which we'll discuss later in this chapter.)

Below is the first case label in SWITCH.C:

```
case 'b':
```

This case label lists the character constant `'b'`. If the variable `ch` equals `'b'`, the program's action branches to this label. If `ch` equals `'\r'`, the program branches to the following label:

```
case '\r':
```

The basic effect of **switch** is similar to the **SELECT CASE** statement in QuickBASIC. The program can branch to many different alternatives, but only one at a time.

A **switch** statement can have as many **case** alternatives as you need. Each alternative must be labeled with a constant value. (You can't use a variable in the label.)

NOTE *In previous versions of QuickC, the constant in a case label could only be a* ***char*** *or* ***int*** *value. In QuickC 2.5, the constant can be any integral type, including a* ***long*** *or* ***unsigned long*** *as well as a* ***char*** *or* ***int****. Chapter 4, "Data Types," describes the* ***char, int,*** *and* ***long*** *types.*

The default keyword is used only in switch statements.

SWITCH.C also shows how to use the **default** keyword in a **switch** statement. The statements after the **default** label execute if the value of the test expression doesn't equal any of the values listed in other labels. In SWITCH.C, the code following **default** executes when the variable `ch` equals anything other than `'b'` or `'\r'`.

Not every **switch** statement requires a **default** label. If no **default** is present, and the test expression doesn't match any of the values listed in the other **case** labels, no statements are executed.

Use the break keyword to exit a switch statement.

You normally place a **break** statement at the end of each alternative, as shown in SWITCH.C. The **break** statement exits the **switch** statement block immediately. If you don't put a **break** at the end of the alternative, the action falls through to the next statement.

For instance, say that you remove all the **break** statements from SWITCH.C, as shown below:

```
switch( ch )
{
   case 'b':
      printf( "Beep!\a\n" );
   case '\r':
      printf( "Enter\n" );
   default:
      printf( "Bye bye" );
}
```

If you run the revised program and type the letter b, the program executes the first alternative, producing this output:

```
Beep!
```

then goes on to execute the statements that follow:

```
Enter
Bye bye
```

Occasionally, you may want to fall through from one **case** alternative to another. But you should be careful not to omit **break** statements accidentally. (See the section "Omitting break Statements from a switch Statement" in Chapter 10, "Programming Pitfalls.")

If you end each alternative with a **break**, as in SWITCH.C, the order of the alternatives isn't critical. The program branches to the label containing the correct value, no matter where that label appears in the **switch** statement block. For instance, you can reverse the order of the alternatives in SWITCH.C without changing the program's output. For readability's sake, many programmers put **default** at the end of a **switch** statement and arrange the other alternatives alphabetically or numerically.

Sometimes you'll want to execute the same code for more than one case. This is done by grouping all the desired labels in front of one alternative. For instance, if you revise the second alternative in SWITCH.C to read

```
case '\r':
case '\t':
case ' ':
   printf( "What a boring choice!\n" );
   break;
```

the program will print

```
What a boring choice!
```

when you press the ENTER key, the TAB key, or the SPACEBAR.

The break Statement

The previous section explained how to use **break** to exit from a **switch** statement. You can also use **break** to end a loop immediately. The BREAKER.C program shows how to do this. The program prints a prompt, then displays characters as they are typed until the TAB key is pressed.

```
/* BREAKER.C: Demonstrate break statement. */

#include <stdio.h>
#include <conio.h>

main()
{
   char ch;
   printf( "Press any key. Press Tab to quit.\n" );
   while( 1 )
   {
      ch = getche();
      if( ch == '\t' )
```

```
            {
               printf( "\a\nYou pressed Tab\n" );
               break;
            }
         }
      }
```

The **while** statement in BREAKER.C creates an indefinite loop that calls the **getche** function again and again, assigning the function's return value to the variable `ch`. The **if** statement in the loop body compares `ch` to the tab character. When TAB is pressed, BREAKER.C prints `You pressed Tab` and executes the **break** statement, which terminates the **while** loop and ends the program.

A break statement exits only one loop.

It's important to remember that the **break** statement only ends the loop in which it appears. If two loops are nested, executing a **break** in the inner loop exits that loop but not the outer loop. BREAKER1.C shows how **break** works within nested loops. The program's inner loop checks for the TAB key and the outer loop checks for the ENTER key.

```
/* BREAKER1.C: Break only exits one loop. */

#include <stdio.h>
#include <conio.h>

main()
{
   char ch;
   printf( "Press any key. Press Enter to quit.\n" );
   do
   {
      while( ( ch = getche() ) != '\r' )
      {
         if( ch == '\t' )
         {
            printf( "\a\nYou pressed Tab\n" );
            break;
         }
      }
   } while( ch != '\r' );
   printf( "\nBye bye." );
}
```

The BREAKER1.C program includes a **while** loop nested within a **do** loop. Both loops test the same condition—whether the variable `ch` equals the ENTER key (**\r**). The **while** loop also calls the **getche** function, assigning the function's return value to `ch`.

When TAB is pressed, the program prints `You pressed Tab` and executes a **break** statement, which terminates the inner loop. The **break** does not end the outer loop, however. The program continues until ENTER is pressed, providing the condition that ends both loops.

Note that **break** can only be used to exit a loop or **switch** statement. While you might be tempted to use **break** to jump out of complex **if** or **else** statements, the **break** statement cannot be used for this purpose. It has no effect on **if** and **else**.

The continue Statement

The continue statement skips remaining statements in the loop body where it appears.

The **continue** statement, like **break**, interrupts the normal flow of execution in a loop body. But instead of ending the loop, **continue** skips all following statements in the loop body and triggers the next iteration of the loop. This effect can be useful within complex loops, in which you might wish to skip to the next loop iteration from various locations.

The CONT.C program shows how **continue** works. It increments the `count` variable, counting from 0 through 9, but stops printing the value of `count` when that value exceeds 3.

```
/* CONT.C: Demonstrate continue statement. */

#include <stdio.h>

main()
{
   int count;
   for( count = 0; count < 10; count = count + 1 )
   {
      if( count > 3 )
         continue;
      printf( "count = %d\n", count );
   }
   printf( "Done!\n" );
}
```

Here's the output from CONT.C:

```
count = 0
count = 1
count = 2
count = 3
Done!
```

The **continue** statement occurs within the body of the **for** loop. When the value of `count` exceeds 3, the **continue** skips the rest of the loop body—a statement that calls **printf**—and causes the next iteration of the loop.

The goto Statement

The goto statement jumps from one part of the function to another.

Similar to the **GOTO** statement in BASIC, **goto** in C performs an unconditional jump from one part of a function to any other part. The target of the **goto** statement is a label which you supply. The label must end with a colon, as do **case** labels, which we discussed earlier.

Most C programmers avoid using the **goto** statement. It's a bit inconsistent with the overall philosophy of C, which encourages structured, modular programming. And, regardless of philosophy, it can be very difficult to read and debug a program that is littered with haphazard unconditional jumps.

Nevertheless, **goto** has at least one sensible use. If a serious error occurs deep within a nested series of loops or conditional statements, **goto** offers the simplest escape. The following code has several levels of nesting, with a **goto** statement at the innermost level. If the value of `error_count` exceeds 15, the **goto** statement executes, transferring control to the label `bail_out`.

```
if( a == 1 )
{
   while( b == 2 )
   {
      for( c = 0; c < 3; c = c + 1 )
      {
         if( d == 4 )
         {
            while( e == 6 )
            {
               if( error_count > 15 )
                  goto bail_out;
            }
         }
      }
   }
}
bail_out:  /* The goto statement transfers control here. */
```

To achieve the same effect without **goto**, you'd have to add extra conditional tests to this code, making the code more complex and perhaps less efficient.

Names in **goto** labels are governed by the rules for variable names, which we'll discuss in the next two chapters. For now, just remember that a **goto** label is visible only in the function in which it appears. You can't execute a **goto** statement to jump from one function to another function.

The next two chapters explain how to create and manipulate data—variables and constants—in C programs. Chapter 4, "Data Types," describes the basics, such as how to declare and initialize variables of different types. Chapter 5, "Advanced Data Types," describes more advanced topics, such as the visibility of variables.

CHAPTER

Data Types

4

This chapter explains the C data types and shows how to declare and use C variables. The chapter begins by describing the basic data types from which all other data types are derived. We then discuss more complex data types, including arrays and structures. In Chapter 5, "Advanced Data Types," we'll explore more advanced data-handling topics, such as variable visibility and automatic type conversions.

Basic Data Types

All data in C programs is either a constant or variable, and each has an associated data type. The concept of types is common to all high-level languages. For instance, an integer (whole) number has the **INTEGER** type in QuickBASIC, the **Integer** type in QuickPascal, and the **int** type in C. This section describes the basic data types in C and explains how to specify variables and constants using these types.

All of the basic data types contain a single value. Types that contain more than one value—arrays, structures, and unions—are called "aggregate types." We'll discuss aggregate types later in this chapter.

Specifying Basic Types

The C language has four basic data types, which are specified with the keywords **char**, **int**, **float**, and **double**. The **char** (character) type is used for text and the **int** type for integers. The **float** and **double** types express real (floating-point) values.

The TYPES.C program creates variables of the four basic types and prints their values:

```
/* TYPES.C: Illustrate basic data types. */

#include <stdio.h>

main()
{
   char char_val       = 'a';
   int int_val         = 543;
   float float_val     = 11.1;
   double double_val   = 66.123456789;
   printf( "char_val   = %c\n", char_val );
   printf( "int_val    = %d\n", int_val );
   printf( "float_val  = %f\n", float_val );
   printf( "double_val = %2.9f\n", double_val );
}
```

Here is the output from TYPES.C:

```
char_val   = a
int_val    = 543
float_val  = 11.100000
double_val = 66.123456789
```

Each basic data type requires a different amount of memory, as illustrated in Figure 4.1. In QuickC, a **char** contains one byte, an **int** has two bytes, a **float** has four bytes, and a **double** type has eight bytes.

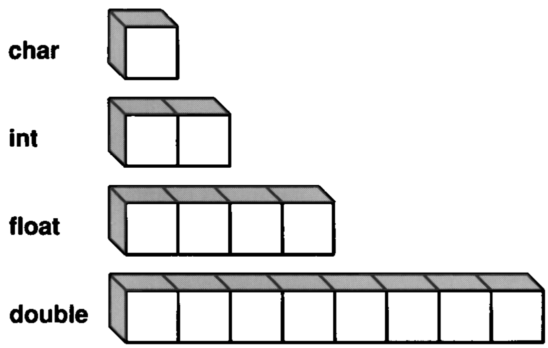


Figure 4.1 Basic Data Types

NOTE *The C language is designed to run on many different computers, with machine architectures that may be quite different. To accommodate these differences, some C data types are "implementation dependent," meaning their sizes depend on which computer you're using. For instance, the* ***int*** *(integer) type contains two bytes on IBM PC computers*

and four bytes on VAX® minicomputers. These differences are important only if you're transporting a program from one operating system to another. Since QuickC runs only under one operating system (DOS), this book describes C data types in DOS.

Special Type Specifiers

The C language has four special type specifiers—**signed**, **unsigned**, **long**, and **short**. These act as "adjectives" to modify the range of values expressed by a basic data type.

The char and int data types are signed by default.

The **signed** keyword signifies that a value can be either negative or nonnegative. If you don't specify, a **char** or **int** value is signed.

You can preface a **char** or **int** with **unsigned** to extend the range of nonnegative values. An **unsigned int** can have a value in the range 0 through 65,535, and an **unsigned char** can have a value of 0 through 255.

The **long** keyword is used to increase the size of an **int** or **double** type. A **long int** value contains four bytes (twice as many as an **int**) and expresses an integer in the range –2,147,483,648 through 2,147,483,647. A **long double** value contains 10 bytes and can express a floating-point number with 19 digits of precision.

In QuickC, the **short int** type is identical to the **int** type. (This is not the case in some operating systems other than DOS.)

Table 4.1 lists the basic data types and the range of values each can express.

Table 4.1 Basic Data Types

Type Name	Other Names	Range of Values
char	**signed char**	–128 to 127
unsigned char	none	0 to 255
int	**signed, signed int**	–32,768 to 32,767
unsigned	**unsigned int**	0 to 65,535
unsigned short	**unsigned short int**	0 to 65,535
short	**short int, signed short signed short int**	–32,768 to 32,767
long	**long int, signed long signed long int**	–2,147,483,648 to 2,147,483,647
unsigned long	**unsigned long int**	0 to 4,294,967,295

Table 4.1 Basic Data Types *(continued)*

Type Name	Other Names	Range of Values
float	none	Approximately 1.2E–38 to 3.4E+38 (7-digit precision)
double	none	Approximately 2.2E–308 to 1.8E+308 (15-digit precision)
long double	none	Approximately 3.4E–4932 to 1.2E+4932 (19-digit precision)

Most programmers take advantage of type defaults. If a type qualifier appears alone, the type **int** is implied. By itself, **short** is a synonym for **short int**. Where **long** appears alone it is a synonym for **long int**, and where **unsigned** appears alone it is a synonym for **unsigned int**.

Specifying Variables

As we mentioned in Chapter 1, "Anatomy of a C Program," you must declare every variable in a C program by stating the variable's name and type. Variable names are governed by the following rules, which also apply to other user-defined names such as function names:

- C is case-sensitive. For example, `myvar`, `MyVar`, and `MYVAR` are different names.
- The name can't be a keyword (see online help for a list of keywords).
- The first character must be a letter or underscore (_). Many of QuickC's system-defined names, including some library-routine names, begin with underscores. To avoid conflicts with such names, don't create names that begin with underscores.
- Other characters can be letters, digits, or underscores.
- The first 31 characters of local variable names are significant. The name can contain more than 31 characters, but QuickC ignores everything beyond the thirty-first character. Global variable names are normally significant to 30 characters.

All C keywords are lowercase, and it's common to use lowercase for variable names. Mixed case is becoming popular in some contexts, however. OS/2 and Microsoft Windows™ use mixed case for most system-defined names.

Specifying Constants

Constants—values that don't change during the life of the program—can be numbers, characters, or strings. Your program can also define "symbolic constants," which are names that represent constant values. This section describes how to specify C constants.

Numeric Constants

A numeric constant can have any basic data type, and can be specified in decimal, hexadecimal, or octal notation. Table 4.2 shows how to specify numeric constants.

Table 4.2 Constant Specifications

Constant	Type
255	decimal **int**
0xFF	hexadecimal **int** (255)
0377	octal **int** (255)
255L	**long int**
255U	**unsigned int**
0xFFul	**long unsigned** hexadecimal **int** (255)
15.75E2	floating point (1575)
–.123	floating point
.123	floating point
3e0f	floating point

A number without a suffix, such as 255, is treated as decimal. The `0x` prefix specifies a hexadecimal number, and the `0` (zero) prefix specifies octal (base 8).

If a number doesn't have a decimal point, it is an integer. Integers are signed by default; you can use the suffix `U` or `u` to specify an unsigned constant. To specify a **long** integer, place the suffix `L` or `l` after the number.

A floating-point constant contains either a decimal point or an exponent preceded by `e` or `E`. It can optionally include the suffix `F` or `f` to denote the **float** type or the suffix `L` or `l` to denote the **long double** type.

Character and String Constants

The C language uses different notation for character and string constants. A single character enclosed in single quotes is a character constant:

```
'a'
```

A string constant is 0 or more characters enclosed in double quotes:

```
"Hello"
```

A string also ends with a null character (**\0**), as we'll see in the section "Strings."

The difference between character and string constants is important when you perform comparisons. The character constant `'a'` contains 1 character, but the string constant `"a"` contains 2 characters: the letter `a` plus a null character. Because the two values have a different number of characters, any comparison of them is invalid. (See "String Problems" in Chapter 10, "Programming Pitfalls.")

You can specify special characters, such as the tab and backspace, with a multi-character sequence that begins with a backslash (\ ). These sequences are sometimes called "escape sequences." Table 4.3 shows the special character sequences.

Table 4.3 Special Characters

Sequence	Character
\a	Alert (bell)
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quote
\"	Double quote
\\	Backslash
ooo	Octal notation
xhh	Hexadecimal notation
\0	Null

Some unusual characters don't have a predefined sequence. You can specify these with a backslash (\ ) followed by the hexadecimal or octal number representing the character's ASCII value. For instance, a telecommunications program

might need to specify ASCII 21, the NAK ("not acknowledged") character. You can specify this character in either hexadecimal notation, `'\x15'`, or octal notation, `'\25'`. Note that the hexadecimal number begins with `\x`, while the octal number starts with a backslash alone.

Symbolic Constants

A "symbolic constant" is a user-defined name that represents a constant. Symbolic constants are usually typed in uppercase. For instance, the directive

```
#define PI 3.14
```

declares a symbolic constant named `PI`. Wherever `PI` occurs in the program, the compiler substitutes `3.14`. Chapter 7, "Preprocessor Directives," discusses symbolic constants and the **#define** directive.

Aggregate Data Types

This section describes aggregate data types, which contain organized collections of data in a definite order. In C, the aggregate types are arrays, structures, and unions.

An "array" is a collection of data items of the same type. Programs use arrays in cases where a standard data format is repeated many times. For example, you might use an array to store numbers representing the population of Minnesota for all the years from 1950 to 2000. C-language arrays are very similar to arrays in QuickPascal and QuickBASIC.

A "structure" is a collection of data items of different types. Programs use structures in cases where a variety of data have a close association. For example, you might use a structure to store information about a given employee—name, months of employment, and hourly wage. Structures are similar to QuickPascal records or QuickBASIC user-defined types.

A "union" allows you to use different data formats to access the same area of memory. It can hold different kinds of information at different times. Unions are similar to variant records in QuickPascal.

Arrays

An array is a group of data items of the same type under one name.

The simplest aggregate data type is an array: a group of data items that share the same type and a common name. You can make an array from any data type, including basic types such as **char** and **int** and more complex types such as structures. This section shows how to declare, initialize, and access arrays, including arrays with more than one dimension. We'll begin with a simple example that creates a one-dimensional array.

Creating a Simple Array

The ARRAY.C program creates the array i_array, which contains three integers.

```
/* ARRAY.C: Demonstrate one-dimensional array. */

#include <stdio.h>

main()
{
   int j;
   int i_array[3];

   i_array[0] = 176;
   i_array[1] = 4069;
   i_array[2] = 303;

   printf( "--- Values --------     --- Addresses -------\n\n" );

   for( j = 0; j < 3; j = j + 1 )
   {
      printf( "i_array[%d] = %d", j, i_array[j] );
      printf( "\t&i_array[%d] = %u\n", j, &i_array[j] );
   }

}
```

Here is the output from ARRAY.C:

```
--- Values --------     --- Addresses -------

i_array[0] = 176        &i_array[0] = 3506
i_array[1] = 4069       &i_array[1] = 3508
i_array[2] = 303        &i_array[2] = 3510
```

As you can see, ARRAY.C prints the values in i_array and the memory address where each array element is stored. You usually don't have to worry about actual memory addresses in C, but it's useful to have some idea how array elements are stored in memory. Depending on factors such as the amount of memory in your system, you may see different addresses when you run ARRAY.C.

(The second **printf** statement uses the "address-of" operator (**&**) to determine the address of each array element. Chapter 6, "Operators," explains this operator. For now, it's sufficient to recognize that the operator allows ARRAY.C to print addresses.)

Figure 4.2 shows how `i_array` is stored in the addresses from the ARRAY.C output.

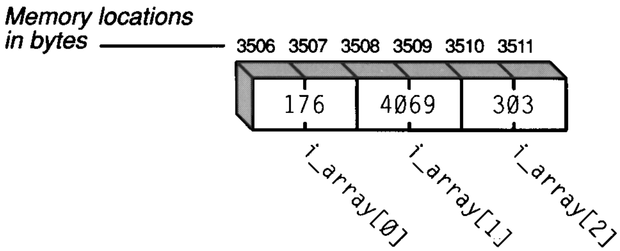


Figure 4.2 Array Storage in ARRAY.C

Declaring the Array

You declare an array variable by stating its type and its name, as you would a simple variable. You must also declare the size of the array, stating the number of elements with an integer constant in square brackets. For example, the line

```
int i_array[3];
```

from ARRAY.C declares a three-element integer (**int**) array named `i_array`.

Multidimensional arrays are declared the same way, except you must give the size of each dimension. The following statement, for instance, declares a two-dimensional **int** array named `two_dim`:

```
int two_dim[2][3];
```

We'll return to multidimensional arrays a little later in this chapter.

Initializing the Array

Arrays, like simple variables, should be initialized before use. ARRAY.C initializes `i_array` with these statements:

```
i_array[Ø] = 176;
i_array[1] = 4Ø69;
i_array[2] = 3Ø3;
```

An array can be initialized when it is declared.

ARRAY.C declares an array in one statement and then initializes its elements one by one. You can also initialize an array when you declare it. The following statement does both jobs at once:

```
int i_array[3] = { 176, 4069, 303 };
```

Note the curly braces around the initializing values. The braces are mandatory in this kind of initialization.

Under the ANSI C standard, which QuickC version 2.5 follows, you can simultaneously declare and initialize an array within a function. Pre-ANSI compilers, including QuickC version 1.0, don't allow this unless the **static** keyword precedes the array declaration. Chapter 5, "Advanced Data Types," discusses **static**.

When you declare and initialize an array at the same time, the initializing values are normally constants, as shown above. Occasionally, you may want to initialize an array as you declare it using variables instead of constants. QuickC version 2.5 allows this, but only within a function. The `sample` array in the following example is initialized legally under QuickC version 2.5 but illegally under QuickC version 1.0:

```
func()
{
   int val = 5;
   int sample[3] = { val, val, val };
}
```

If you initialize a local array in this way, you must include the size of the array within the square brackets following the array name. If the example initialized the `sample` array with the following line:

```
int sample[ ] = {val, val, val};
```

QuickC would issue an error because the size of the array (3) is not specified.

Specifying Array Elements

Array subscripts are enclosed in square brackets ([ ]).

You specify an array element by giving its position, using an integer value called a "subscript." Square brackets ([ ]) enclose each subscript. In the ARRAY.C program above we specify the first element of `i_array` as

```
i_array[0]
```

Notice that the first element of a C array has the subscript 0, not 1. Unlike QuickPascal and QuickBASIC, the C language does not give you the option to start at an index number other than 0.

Since array subscripts begin at 0, the subscript of the last array element is 1 less than the number used to declare that dimension of the array. In ARRAY.C, the last element of `i_array` is `i_array[2]`, not `i_array[3]`.

C doesn't check array subscripts.

Unlike QuickBASIC and QuickPascal, C doesn't check the validity of array subscripts. If the ARRAY.C program included the expression

```
i_array[55];
```

it would refer to a nonexistent array element. (The expression refers to the element 55, but `i_array` contains only three elements.) This would not trigger a compiler error or run-time error, however. It's your job to remember the size of the array and avoid references that go outside the array's boundaries. This rule is also important when you're accessing arrays with pointers (see Chapter 8, "Pointers").

Strings

A string is an array of characters.

You may have wondered why we didn't mention strings in our earlier description of basic data types. The reason is that strings aren't a formal data type. In the C language, a string is simply an array of characters (**char** values).

The STRING.C program below creates the string `c_array` and displays its contents in the same format as the previous example. The program prints the value of each array element and its address.

```
/* STRING.C: Demonstrate string arrays. */

#include <stdio.h>

main()
{
   int j;
   char c_array[] = "Hello";

   printf( "--- Values                    Addresses -------\n\n" );

   for( j = 0; j < 6; j = j + 1 )
   {
      printf( "c_array[%d]   = %x %c", j, c_array[j], c_array[j] );
      printf( "\t&c_array[%d]    = %u\n", j, &c_array[j] );
   }
}
```

Here is the output from STRING.C:

```
    Values              --- Addresses -------

c_array[0]    = 48 H    &c_array[0]    = 3522
c_array[1]    = 65 e    &c_array[1]    = 3523
c_array[2]    = 6c l    &c_array[2]    = 3524
c_array[3]    = 6c l    &c_array[3]    = 3525
c_array[4]    = 6f o    &c_array[4]    = 3526
c_array[5]    = 0       &c_array[5]    = 3527
```

Figure 4.3 shows how `c_array` is stored in memory. Again, the addresses in the output may differ depending on factors such as the amount of available memory.

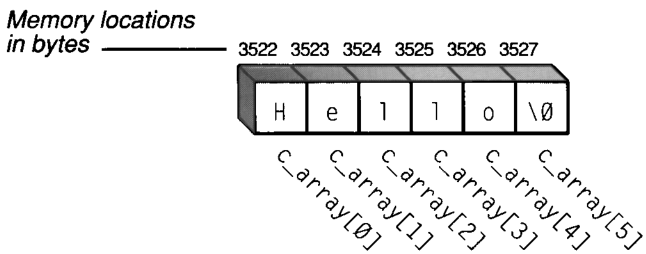


Figure 4.3 String Storage in STRING. C

A string ends with a null character.

The figure illustrates another important feature of strings. Although `c_array` has five printing characters (`Hello`), it actually contains six characters—five letters plus a null character (**\0**) that marks the end of the string. As noted earlier, the C language automatically adds a null character to every string enclosed in double quotes.

STRING.C uses a shortcut when it initializes `c_array`. You may have noticed that the array declaration

```
char c_array[] = "Hello";
```

doesn't declare the array's size (the square brackets are empty). When an array is initialized at the same time it's declared, QuickC can figure out how many elements the array has by counting the number of initializing values to the right of the equal sign.

You can use this shortcut for any type of array, not just a **char** array. If the array has more than one dimension, however, you can only omit the size of the first dimension.

Multidimensional Arrays

A "multidimensional" array contains two or more array dimensions. The TWODIM.C program below creates a two-dimensional array named `i_array`.

```
/* TWODIM.C: Demonstrate multidimensional arrays. */

#include <stdio.h>

main()
{
   int j, k;
   int i_array[2][3] = { { 176, 4069, 303 }, { 6, 55, 777 } };

   printf( "--- Values --------     --- Addresses -------\n\n" );

   for( j = 0; j < 2; j = j + 1 )
   {
      for( k = 0; k < 3; k = k + 1 )
      {
         printf( "i_array[%d][%d] = %d", j, k, i_array[j][k] );
         printf( "\t&i_array[%d][%d] = %u\n", j, k, &i_array[j][k] );
      }
      printf( "\n" );
   }

}
```

Here's the output from TWODIM.C:

```
--- Values --------     --- Addresses -------

i_array[0][0] = 176     &i_array[0][0] = 3498
i_array[0][1] = 4069    &i_array[0][1] = 3500
i_array[0][2] = 303     &i_array[0][2] = 3502

i_array[1][0] = 6       &i_array[1][0] = 3504
i_array[1][1] = 55      &i_array[1][1] = 3506
i_array[1][2] = 777     &i_array[1][2] = 3508
```

Each subscript of a multidimensional array appears in its own set of square brackets, as the TWODIM.C output shows. When you declare the array, the first subscript states the size of the first dimension, the second states the size of the second dimension, and so on. In TWODIM.C, the declaration of `i_array`,

```
int i_array[2][3]
```

states that `i_array` contains two rows of values, each row containing three integers. The statement that declares `i_array` also initializes the array, listing the initializing values in curly braces to the right of the equal sign:

```
int i_array[2][3] = { { 176, 4069, 303 }, { 6, 5, 77 } };
```

The braces clearly show that the array contains two groups of three values.

Two-dimensional arrays are often pictured in rows and columns, as in Figure 4.4. Of course, since computer memory is linear, `i_array` is actually stored with its rows end-for-end, as in Figure 4.5.

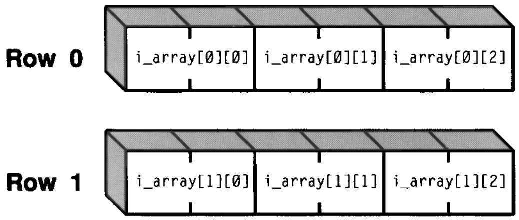


Figure 4.4 Two-dimensional Array

You refer to a multidimensional array element the same way you would a one-dimensional array element, except that you use one subscript for each dimension of the array. For instance, the statement

```
printf( "%d\n", i_array[0][1] );
```

specifies two subscripts. It prints the value stored in element 0, 1 of `i_array`, which is `4069`.

Figure 4.5 shows how to specify every element of `i_array` in TWODIM.C.

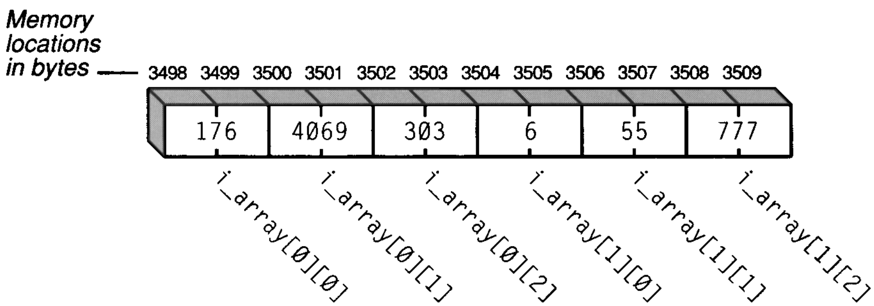


Figure 4.5 Array Storage in TWODIM.C

Structures

A structure is a group of related data items of different types under one name.

The second aggregate data type is the structure: a group of related data items under one name. While array elements are all the same type, the elements of a structure, known as its "members," can be of different types.

Structures are equivalent to records in QuickPascal or user-defined types in QuickBASIC. As in those languages, the ability to group different types in the same construct provides powerful, very flexible data-handling capabilities.

Creating a Simple Structure

We'll write a simple program to demonstrate the basics of structures. Suppose you want to write a payroll program that records these facts about an employee:

- Name
- Number of months of service
- Hourly wage

Each of these data items requires a different data type. The name can be stored in a string (character array), while an integer will do for the months of service. The hourly wage may contain a fraction; we'll store it in a floating-point variable.

Although each of these variables has a different type, we can group all of them in a single structure. The EMPLOYEE.C program below contains the structure.

```
/* EMPLOYEE.C: Demonstrate structures. */

#include <stdio.h>
#include <string.h>

struct employee
{
   char name[10];
   int months;
   float wage;
};

void display( struct employee show );

main()
{
   struct employee jones;

   strcpy( jones.name, "Jones, J" );
   jones.months = 77;
   jones.wage = 13.68;

   display( jones );
}

void display( struct employee show )
{
   printf( "Name: %s\n", show.name );
```

```
    printf( "Months of service: %d\n", show.months );
    printf( "Hourly wage: %6.2f\n", show.wage );
}
```

Here is the output of the EMPLOYEE.C program:

```
Name: Jones, J
Months of service: 77
Hourly wage:  13.68
```

Declaring a Structure Type

Since a structure can (and normally does) contain different data types, creating it is a little more complicated than making an array or simple variable. Before you can create a structure variable, you must declare a structure type that tells the compiler how many members the structure contains and what types they are.

A structure-type declaration starts with the keyword **struct**, which is followed by a list of the structure's members enclosed in braces. Between the **struct** and the list of members, you can also specify a "structure tag"—a name that other parts of the program can use to refer to the type.

The structure declaration from EMPLOYEE.C,

```
struct employee
{
   char name[10];
   int months;
   float wage;
};
```

A structure declaration makes a template for variables of the type it defines.

creates a "template" for an `employee` structure that structure variables of this type can use. It's as if you created a brand new data type, tagging it `employee`. Figure 4.6 illustrates the `employee` structure type.

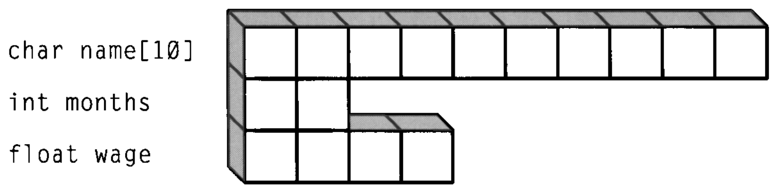


Figure 4.6 The employee Structure Type

Creating a Structure Variable

Once you have declared a structure, you can create variables of that type using the structure tag. Each variable can contain values of the types defined in the structure type. In EMPLOYEE.C, the statement

```
struct employee jones;
```

declares a structure variable of the type `employee` named `jones`. The **struct** states that the variable is a structure. The `employee` tag specifies the variable's structure type, and `jones` is the variable's name.

You can also declare the variable in the same statement that declares the structure type. The following code declares the `employee` structure type and a variable of that type named `jones`:

```
struct employee
{
   char name[10];
   int months;
   float wage;
} jones;
```

The variable name (`jones`) appears at the end of the declaration.

Use the member-of operator (.) to specify structure members.

You specify structure members by name, using the "member-of" operator (**.**) to separate the variable name and the member name. These are the names of the members of the `jones` structure variable in EMPLOYEE.C:

```
jones.name
jones.months
jones.wage
```

Like other variables, structure variables should be initialized before use. After `jones` is declared in EMPLOYEE.C, the statements

```
strcpy( jones.name, "Jones, J" );
jones.months = 77;
jones.wage = 13.68;
```

initialize the members of the `jones` variable. The first statement initializes the `jones.name` member by calling the **strcpy** ("string copy") library function; this function is described in Chapter 11, "Input and Output."

Figure 4.7 shows how the `jones` structure is stored in memory. Again, since computer memory is linear, the members of the structure are laid out end-to-end.

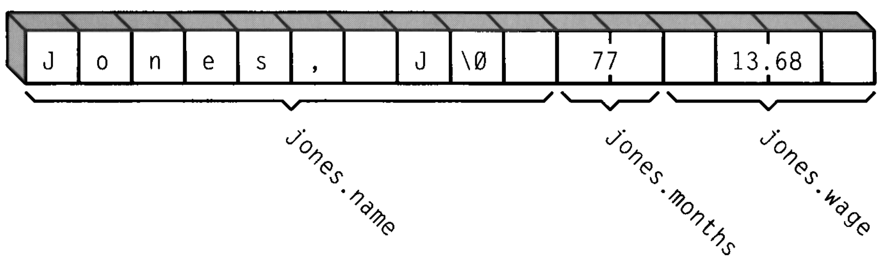


Figure 4.7 Structure Storage in EMPLOYEE.C

You can initialize a structure when you declare it. The following code would perform both operations in EMPLOYEE.C:

```
struct employee jones =
{
   "Jones, J",
   77,
   13.68
};
```

This code declares the `jones` structure variable and lists the initializing value for each of its members.

Using Structure Variables

A structure member can be treated like any other variable of its type. You can assign a value to it, change its value, and so on. For instance, the statement

```
jones.months = 83;
```

would change the value of the `jones.months` member in EMPLOYEE.C.

Assigning one structure to another copies the entire structure.

You can also assign an entire structure to another structure of the same type. This copies the entire contents of the first structure to the second. You might do this to save time when creating a new structure whose contents differ only slightly from those of an existing structure.

To illustrate, let's modify the EMPLOYEE.C program. Say you have a second employee named Lavik whose wage rate and months of service are the same as those of Jones and you want to create a second structure. You could begin by declaring a second `employee` structure variable named `lavik` in this fashion:

```
struct employee lavik = jones;
```

Now the members of the `lavik` structure contain the same data as the members of the `jones` structure. The `lavik.name` member contains the string `Jones, J`, the `lavik.months` member contains the value 77, and the `lavik.wage` member contains the value 13.68. You could add the statement

```
strcpy( lavik.name, "Lavik, B" );
```

to place a new string in the `lavik.name` member.

Structure variables can be passed as function arguments.

When you pass a structure name to a function, the function creates a local structure variable of that type. Like all local variables, the new variable is private to the function that includes it.

For example, if you add the statements

```
strcpy( show.name, "King, M" );
printf( "%s\n", show.name );
```

to the end of the `display` function in EMPLOYEE.C, then a new string is copied into the `show.name` member of the function's structure variable. The **printf** statement in the second line prints

```
King, M
```

Since this structure is local to the `display` function, the change doesn't affect the structure defined in the **main** function. If you add the statement

```
printf( "%s\n", jones.name );
```

to the end of the **main** function, the program prints

```
Jones, J
```

The original structure is unchanged.

While you can pass a structure name to a function as we did above, it's more common to pass the function a *pointer* to the structure. This not only permits the function to access a structure defined elsewhere in the program, but it conserves memory (since the function doesn't create a local copy of the structure). Chapter 9, "Advanced Pointers," explains how to access structures using pointers.

Arrays of Structures

An array of structures is a group of structures of the same type.

Since it's rare for a company to have a single employee, a more practical version of the EMPLOYEE.C program would have an array of structures—one structure per employee. The concept may sound intimidating, but this is a common use of structures.

The following statement declares a 50-element array named `payroll`, with each element a structure of the type `employee` :

```
struct employee payroll[50];
```

To specify members in such an array, you combine array notation and structure notation, giving the array name, a subscript, and a member name. For instance, the name

```
payroll[0].months
```

specifies the `months` member of the first structure in the `payroll` array. The first part of the name (`payroll[0]`) contains the array name and subscript that identify the structure. The second part (`months`) identifies the member within that structure.

Figure 4.8 depicts the first three elements of the `payroll` array.

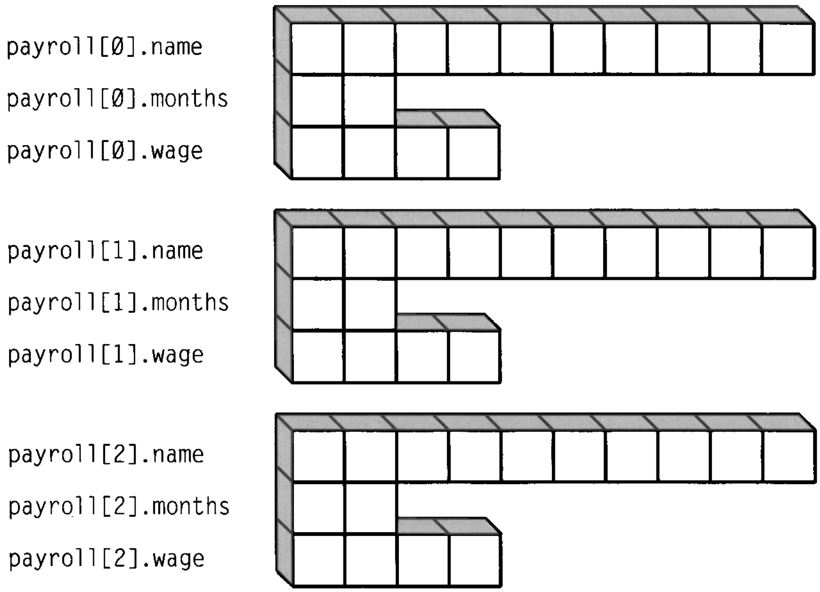


Figure 4.8 Array of Structures

Once you grasp the basic idea, it's easy to imagine practical uses for an array of structures. Many programs, from an address book to a library card catalog, might use a structure to store different types of information about an individual item, then store many such structures in an array.

Structures of Structures

As noted earlier, a structure can contain members of any data type—including other structures. So you can create a structure of structures: a structure whose members are structures.

To illustrate, suppose you write a group of functions that draw various kinds of graphic windows and message boxes. You could define a small structure something like the following:

```
struct title
{
   char text[70];  /* Title text */
   int color;      /* Color of title text */
   short justify;  /* Left, center, or right */
};
```

to aid in drawing titles. The `title` structure's three members specify the title's text, its color, and how its text is justified.

Once the `title` structure is defined, you can make it part of other, larger structures that use titles. If you define a `window` structure type to draw windows, for example, that structure could include a `title` along with other structure members:

```
struct window
{
   struct title wintitle; /* Window title */
   /* Other structure members go here... */
};
```

In this structure type, the `title` member is named `wintitle`.

You specify members of such structures using member-of operators and the appropriate names. If you create a variable of the `window` type named `mywindow`, the name

```
mywindow.wintitle.color
```

specifies the `color` member of the `wintitle` member of the `mywindow` structure.

If you program using QuickC's Presentation Graphics library, you'll find it useful to understand the notation we just explained. Our fictitious `title` structure is a simplified version of the Presentation Graphics **titletype** structure type (see Chapter 14, "Presentation Graphics").

Bit Fields

A "bit field" is a specialized structure that provides a way to manipulate individual bits or groups of bits. One use for this advanced feature is to access hardware addresses such as the computer's video memory.

The members of a bit-field structure are groups of bits.

You declare and use a bit-field structure much as you would any other structure. The difference is that every one of its members must be a bit or group of bits. You can't include other data types in a bit field.

The following statement declares a bit-field structure type with the tag `SCREEN`:

```
struct SCREEN
{
  unsigned character   8;
  unsigned fgcolor     3;
  unsigned intensity   1;
  unsigned bgcolor     3;
  unsigned blink       1;
} screenbuf[25][80];
```

The colons in the declaration tell QuickC these are bit fields rather than normal structure members. The number following each colon tells how many bits the

field contains. In the `SCREEN` type the `character` member has 8 bits, `intensity` has 1 bit, and so on. Figure 4.9 illustrates the `SCREEN` type.

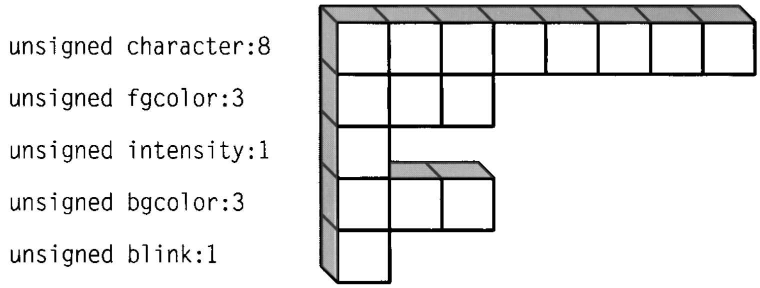


Figure 4.9 The SCREEN Bitfield Structure (Memory Units in Bits)

Figure 4.10 illustrates memory allocation for the `SCREEN` type. The members of the `SCREEN` type mirror the arrangement of bits in screen memory.

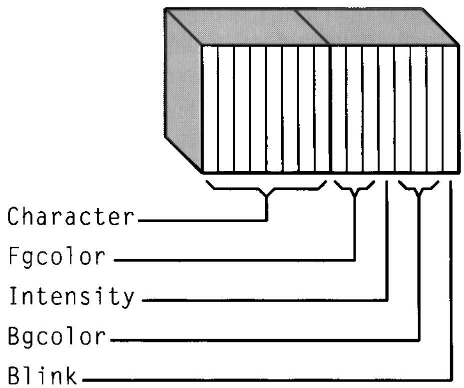


Figure 4.10 Bitfield Structure Storage in Memory

Take another look at the structure declaration. In addition to declaring a structure type, the statement declares a two-dimensional array variable `screenbuf`, of the same structure type. You could use this array as an alternate video buffer. Many graphics programs use a similar arrangement to switch between an alternate video buffer and the computer's video memory.

The five members of the `SCREEN` type happen to take up a full **int** (two bytes, on DOS machines). A bit field need not fill up a byte or **int**; the bit field can contain as many bits as you need up to the maximum number of bits for the field's

base type. The base type for each field in the example is **unsigned** (**unsigned int**), so each field can contain a maximum of 16 bits.

The members of a bit-field structure are accessed with the structure-member operator—like other structure members. For instance, the name

```
screenbuf[13][53].blink = 1;
```

specifies the `blink` member of element 13, 53 of the `screenbuf` array.

The range of values you can assign to a bit-field member depends on the member's size. Since the `blink` member of the `SCREEN` type contains one bit, `blink` is limited to the value 0 or 1. The `fgcolor` member contains three bits and can have any value from 0–7.

Unions

A union is a group of variables of different types that share storage space.

A union is a variable that can hold any one of several data types at different times, using the same storage space. Unions are a rather advanced feature. One use of them is to access DOS registers, which you may sometimes need to access as bytes and at other times as words.

As with a structure, you must start by declaring a union type to tell the compiler the number and types of the union's members. You include one of each type that you expect to use.

The following code creates a union that can hold a **char**, **int**, or **long** value. It declares a union type with the tag `u_sample` and declares a variable of that type named `example`.

```
union u_sample
{
   char c_val;
   int i_val;
   long l_val;
}  example;
```

When you declare a union, QuickC allocates as much storage as the largest data type in the union requires. Since the largest type in `u_sample` is **long**, this union contains four bytes.

The elements of a union are called members and use the same notation as structure members. Thus, the members of the `example` union are named

```
example.c_val
example.i_val
example.l_val
```

The contents of a union depend on how you access it. For instance, the statement

```
example.c_val = '\0';
```

stores a **char** value in the `example` union. Since a **char** value takes one byte, the statement uses only one byte of the space in `example`. The statement

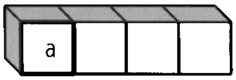
```
example.i_val = 77;
```

uses two bytes of the union, because an **int** value requires two bytes of storage. Likewise, the statement

```
example.l_val = 75621;
```

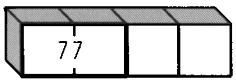
stores **long** value in `example`, taking up all four bytes of its storage space. Figure 4.11 shows memory allocation for the three members in the `example` union.

Storing a char Value in the example Union



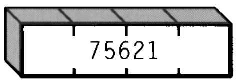

```
example.c_val = 'Ø';
```

Storing an int Value in the example Union




```
example.i_val = 77;
```

Storing a long Value in the example Union




```
example.l_val = 75621;
```

Figure 4.11 Storage in the example Union

It's your job to keep track of what is stored in a union. If you store a **long** value in `example` and mistakenly treat that value as a **char** value later, the result may be nonsense. It's especially important not to confuse integer and floating-point types, which are stored in different internal formats.

Now that you're familiar with the data types that C offers, you are ready to tackle more advanced data-handling concepts. The next chapter discusses several of these.

CHAPTER 5

Advanced Data Types

In Chapter 4, "Data Types," we described the basic C data types and showed how to declare and use different kinds of variables. This chapter examines more advanced data topics, including the visibility and lifetime of variables and the conversion of values from one data type to another.

If you know QuickPascal or QuickBASIC, some of these topics, such as visibility, should be familiar. For example, a variable declared within a function is visible (accessible) only in that function. One area in which C differs notably from QuickPascal is type conversion. The C language gives programmers the freedom to convert a value from one type to another type, whereas QuickPascal does not.

Visibility

Every variable in a C program has a definite "visibility" that determines which parts of the program can "see," or access, the variable. Another term for visibility is "scope."

As we mentioned in Chapter 1, "Anatomy of a C Program," there are two basic kinds of visibility: local and external. A "local" variable—one declared within a function—is visible only within that function. An "external" variable—one declared outside all functions—is visible to all functions that follow it in the program.

This section begins by describing local and external visibility, then goes on to discuss visibility in multiple-file programs and the visibility of functions.

NOTE *While the examples in this section use simple **int** variables, visibility rules apply equally to aggregate types such as arrays and structures.*

Use external variables only when necessary.

C programmers normally limit the visibility of each variable to those parts of the program that need to access the variable. For instance, if a variable is needed only within one function, it should always be local to that function. By restricting a variable's visibility, you can prevent other parts of the program from accidentally changing the variable's value. Such haphazard side effects were common in older interpreted BASIC programs, in which every variable had unlimited visibility.

Local Variables

Variables declared within a function are local to that function.

As we noted in Chapter 1, "Anatomy of a C Program," and Chapter 2, "Functions," the place where you declare a variable controls where the variable is visible. Declaring a variable within a function makes the variable local to that function; the variable can be seen only within the function.

The VISIBLE.C program below demonstrates local visibility. It contains a function named `be_bop` that tries to print the value of the local variable `val`.

```
/* VISIBLE.C: Demonstrate local visibility. */

#include <stdio.h>
void be_bop( void );

main()
{
   int val = 10;
   be_bop();
}

void be_bop( void )
{
   printf( "val = %d", val ); /* Error! */
}
```

Notice where the `val` variable is declared. The declaration

```
int val = 10;
```

occurs within the **main** function, so `val` is local to **main**. When you compile VISIBLE.C, QuickC stops, providing this error message:

```
C2065: 'val'  undefined
```

What happened? The **printf** statement in the `be_bop` function

```
printf( "val = %d", val ); /* Error! */
```

can't "see" the variable `val`, which is declared locally within **main**. Outside the **main** function, in which `val` is declared, the variable doesn't exist.

You could eliminate the error message by declaring `val` externally, but most programmers would avoid that solution. If a variable has external visibility, any part of the program might change its value accidentally. A better solution is to pass the value of `val` to `be_bop` as a function argument, as shown in the program VISIBLE1.C below.

```
/* VISIBLE1.C: Demonstrate local visibility. */

#include <stdio.h>

void be_bop( int param );

main()
{
   int val = 10;
   be_bop( val );
}

void be_bop( int param )
{
   printf( "%d\n", param );
}
```

The VISIBLE1.C program is identical to VISIBLE.C except for two changes. The `be_bop` function now can accept an argument, and the statement that calls `be_bop` passes the value of `val` as an argument. These changes allow the `be_bop` function to print the value of `val` without the drawback of making `val` external.

Most local variables are declared at the beginning of the function and are visible throughout the function. If you declare the variable later in the function, it is visible only to statements that follow the declaration.

The reason for this rule is simple: QuickC, like all language compilers, reads your program line by line, from beginning to end. Until the compiler sees the variable's declaration, it must treat the variable as undefined. This rule applies to all variables, including external variables, as we'll see in the next section.

Although the practice isn't common, you can restrict a local variable's visibility even further by declaring it in a statement block inside a function. For instance, you might declare a variable within the body of the loop or conditional statement. In fact, any pair of curly braces limits the visibility of a variable declared within that pair.

External Variables

If you declare a variable outside all functions, the variable has external visibility; every function that follows the declaration can see the variable. External variables are called "global" in some other languages.

Experienced C programmers use external variables only when necessary—for instance, when two or more functions need the ability to change the same variable or communicate with each other by changing a variable. Even in those cases, however, you may be able to avoid the dangers of external visibility by passing a pointer to the variable as a function argument. See the section "Passing Pointers to Functions" in Chapter 8 ("Pointers") for more information.

Most external variables are declared near the beginning of the program, before any function definitions. In this way, you can make the variable visible to every function in the program. You could do this in VISIBLE1.C by placing the declaration of `val`,

```
int val = 10;
```

immediately before the **main** function.

If you declare the variable `val` later in the program, it is not visible to functions that precede the declaration. The VISIBLE2.C program below demonstrates this principle.

```
/* VISIBLE2.C: Demonstrate external visibility. */

#include <stdio.h>

void be_bop( int param );

main()
{
   be_bop( val ); /* Error! */
}

int val = 10;

void be_bop( int param )
{
   printf( "val = %d\n", param );
}
```

The VISIBLE2.C program is identical to VISIBLE1.C except that `val` is declared externally

```
int val = 10;
```

following the **main** function, rather than locally within **main**.

Because the declaration occurs outside all functions, the variable is external. However, because the declaration follows the **main** function, the variable is not visible within **main**. When the **printf** statement in the **main** function refers to `val`, QuickC issues the error message:

```
C2065: 'val'  undefined
```

Remember, QuickC reads the program line by line, from start to finish. Since the compiler knows nothing about `val` when it reaches the reference in **main**, it must treat `val` as undefined. In this program, only the `be_bop` function can refer to `val`.

Visibility in Multiple Source Files

A "source file" is the file containing your program's text. Source files normally have the .C file extension, to distinguish them from other files such as executable (.EXE) files.

Simple programs have only one source file, but large programs are often split into several source files. If you write a word-processing program, for instance, you might place all the program's screen-output functions in one file, all the file-handling functions in a second file, and so forth.

Use the extern keyword to make an external variable visible in more than one source file.

Normally, an external variable is visible only in the source file in which it is declared. In a multi-file program, however, a function in one file might need to access a variable in a second file. To make the variable visible in more than one source file, you must declare it with the **extern** keyword.

Let's look at a short two-file program that shows how to use **extern**. The first source file, FILE1.C, declares two external variables, `chico` and `harpo`. The file contains one function (**main**) that calls a second function named `yonder`.

```
/* FILE1.C: Visibility in multiple source files. */

int chico = 20, harpo = 30;
extern void yonder( void );

main()
{
   yonder();
}
```

The second source file, FILE2.C, contains the `yonder` function that is called in FILE1.C. This file also declares the variables `chico` and `harpo`, but it prefaces their declarations with **extern** to show that the variables are defined externally in some other file. Once this is done, any function in FILE2.C can refer to `chico` and `harpo` as if they are defined in the same file.

```
/* FILE2.C: Visibility in multiple source files. */

#include <stdio.h>

void yonder( void )
{
   extern int chico, harpo;
   printf( "chico = %d, harpo = %d\n", chico, harpo );
}
```

You can compile this program in one of two ways. In the QuickC environment, choose Set Program List from the Make menu and add FILE1.C and FILE2.C to the list. Then choose Build Program from the Make menu.

You can also enter this command from the DOS command line:

```
qcl FILE1.C FILE2.C
```

In either case, the executable file is named FILE1.EXE. The program's output,

```
chico = 20, harpo = 30
```

shows that the `yonder` function in FILE2.C can access the variables defined in FILE1.C.

Sometimes you may want an external variable to be visible only in the source file where it's declared. The variable can be shared by functions in one file, but it is hidden to all other files, thus minimizing the risk of naming conflicts.

The static keyword can limit a variable's visibility to one source file.

To limit a variable's visibility to one file, precede the variable's declaration with the keyword **static**. For example, if FILE1.C declared the `harpo` variable as **static** in this manner,

```
static int harpo;
```

it would prevent FILE2.C from accessing `harpo` at all, even though FILE2.C declares (with **extern**) that `harpo` is defined somewhere else.

Visibility of Functions

Functions are normally visible in multiple source files.

Unlike variables, functions are external by default. That is, they are normally visible to every file in a multi-file program. You'll notice that in FILE1.C we declared the `yonder` function with the **extern** keyword. We did this merely to improve readability; the keyword shows clearly that the function is defined in some other file. If we removed the **extern** from the declaration of `yonder` in FILE1.C, the program would work just as well as before.

At times you may want to restrict the visibility of a function in a multi-file program, making it visible in some files but not in others. By "hiding" a function from other parts of a program, you can reduce the danger of naming conflicts. For instance, if you write a library of functions to sell commercially, you probably would hide all of the library's local function names, to prevent conflicts with function names your customers might create.

The static keyword can limit a function's visibility.

As with external variables, you limit a function's visibility using the **static** keyword. A function declared as **static** is visible only in the source file that declares it. If we add **static** to the header of the `yonder` function, for example,

```
static void yonder( void )
```

the function could no longer be called from the FILE1.C file.

Lifetime

In addition to visibility, every variable also has a certain "lifetime"—that is, the period during the program's execution when the variable exists.

External variables exist for the life of the program. Memory is allocated for them when the program begins and remains until the program ends.

An automatic variable disappears when the function ends.

Local variables have shorter lifetimes. They come into being when the function begins and disappear when the function ends. For this reason, a local variable is said to be "automatic." The variable comes and goes automatically, each time the function is called.

Automatic variables conserve memory in a couple of ways. First, since they evaporate when the function ends, automatic variables don't consume memory when not in use. Second, they are stored in the "stack" memory area, which the program allocates at run time. So, automatic variables don't enlarge the executable program.

The C language provides the **auto** keyword for declaring automatic variables. However, this keyword is seldom used, since all local variables are automatic unless you specify otherwise. In the following function, both `val` and `example` are automatic variables:

```
void sample( void )
{
   int val;
   auto int example;
}
```

The **auto** preceding the declaration of `example` has no practical effect. The variable `example` is automatic even if you remove the **auto** from its declaration.

Extending the Lives of Local Variables

Occasionally, you may want a local variable to retain its value between function calls. The **static** keyword, introduced earlier as a means of limiting the visibility of external variables, also performs this task.

A static local variable retains its value through subsequent function calls.

If you precede a local variable declaration with **static**, the variable exists for the life of the program—the same lifetime as an external variable. The variable still has local visibility, however.

The STATIC.C program below shows how to create and use a **static** local variable. In STATIC.C, the value of the `methuselah` variable persists through all calls to the `add_val` function, which adds values to `methuselah` and prints the variable's value.

```
/* STATIC.C: Demonstrate static variables. */

#include <stdio.h>

void add_val( int value );

main()
{
   add_val( 1 );
   add_val( 5 );
   add_val( 20 );
}
```

```
void add_val( int value )
{
   static int methuselah;
   if( value == 1 )
      methuselah = 0;
   methuselah = methuselah + value;
   printf( "methuselah = %d\n", methuselah );
}
```

The `add_val` function in STATIC.C accepts one parameter and also declares a **static** local variable named `methuselah`. Each time `add_val` is called, it adds the passed value to `methuselah`.

The **main** function calls the `add_val` function three times, passing the values 1, 5, and 20 to `add_val` as arguments. The program's output

```
methuselah = 1
methuselah = 6
methuselah = 26
```

shows that the value of `methuselah` persists through all three function calls.

If we remove the **static** keyword from the declaration of `methuselah`, the variable's value is not preserved between function calls. The value of `methuselah` is unpredictable the second and third times that `add_val` is called.

Notice that extending a local variable's lifetime with **static** doesn't affect its visibility. The `methuselah` variable keeps its value between function calls, but you can't refer to the variable outside the `add_val` function.

Converting Data Types

It's usually best to avoid mixing data items of different types in the same expression. You wouldn't normally add a character variable to a floating-point variable, for instance. Some languages, such as QuickPascal, generally treat type mixing as an error. However, the C language gives you the freedom to mix data types when necessary.

For example, since the **char** and **int** types both can store whole numbers, there may be times when you have a good reason to add a **char** value to an **int** value. When you mix types, QuickC does not issue an error message. Instead, the compiler converts both data items to the same type and then performs the requested operation.

Type conversion can occur in one of two ways. The first way occurs automatically when you combine different types in an expression. You can also use special syntax to intentionally "cast" (convert) one type to another. We'll discuss both methods in the following sections.

Knowing how C converts types will help you to find bugs that result from unintended type clashes and to minimize errors when you deliberately mix types.

Ranking of Data Types

For purposes of conversion, the C language ranks data types in the order shown in Figure 5.1.

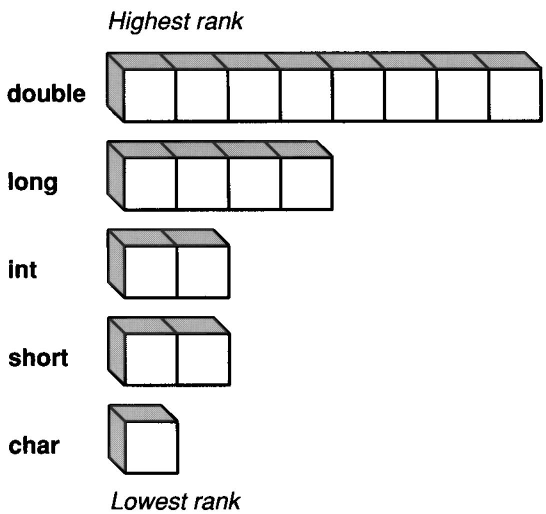


Figure 5.1 Ranking of C Data Types

The ranking illustrated in Figure 5.1 generally reflects the amount of storage that each type requires. As you may remember from Chapter 4, "Data Types," larger data types require more storage than smaller types. Thus, an **int**, which requires two bytes of storage, outranks a **char**, which requires one byte.

Within this ranking, an **unsigned** type outranks the corresponding **signed** type. An **unsigned char** value is of higher rank than a **signed char**, and so forth.

Promotions and Demotions

A promotion is usually harmless.

A type conversion always involves two data items of different types. Whenever possible, QuickC converts the lower-ranking (smaller) data item to the higher-ranking (larger) type. This kind of conversion, called a "promotion," is normally harmless. For example, since a two-byte **int** has more than enough room to store a one-byte **char**, it's generally safe to promote a **char** value to an **int**.

A demotion usually causes a loss of data.

Sometimes, the compiler is forced to convert a higher-ranking value to a lower-ranking type. This kind of conversion, called a "demotion," usually causes loss of data. For example, the **int** value 32,000 is much too large to be stored in a **char** type, which can't hold a number larger than 255. If you assign the value 32,000 to a **char** variable, some data must be lost.

A demotion of an integer type truncates the higher-ranking type, throwing away the data from high-order bytes that can't fit in the smaller-ranking value. Some demotions of floating-point types round off a value rather than truncate it.

Automatic Type Conversions

C does an automatic type conversion when you mix different data types.

When a program statement mixes two different data types, QuickC performs an automatic type conversion. The following code, for instance, adds the **char** variable `a` to the **int** variable `b`.

```
char a = 5;
int b = 32000;
b = a + b;
```

In the statement

```
b = a + b;
```

the addition operation to the right of the equal sign triggers an automatic type conversion. QuickC promotes the **char** value to an **int** and then adds the two **int** values.

If you're not sure whether QuickC is doing an automatic type conversion, set Warning Level 2 or higher in the Compiler Flags dialog box. The compiler generates the warning message

```
C4051: data conversion
```

whenever an automatic conversion occurs. This monitoring helps you readily identify unwanted conversions.

If you carelessly mix different types, you may create subtle errors. The CONVERT.C program below has a deliberate error that shows what can happen when types are mixed. It adds four variables and assigns their sum to a fifth variable, causing three promotions and one demotion.

```
/* CONVERT.C: Demonstrate type conversions. */

#include <stdio.h>

main()
{
   char c_val = 10;
   int i_val = 20;
   long l_val = 64000;
   float f_val = 3.1;
   int result;

   result = c_val + i_val + l_val + f_val;  /* Error! */

   printf( "%d\n", result );
}
```

The CONVERT.C program adds the numbers 10, 20, 64000, and 3.1. Instead of the correct result, 64033.10, the program prints

```
-1503
```

Something definitely went wrong. The problem lies somewhere in the line

```
result = c_val + i_val + l_val + f_val;
```

which triggers four automatic type conversions. We'll examine the conversions in order.

The first conversion occurs when the **char** variable `c_val` is added to the **int** variable `i_val`:

```
c_val + i_val
```

Since the variables are different types, QuickC automatically converts the lower-ranking **char** value to the higher-ranking **int** type before adding them. This promotion doesn't create any problems, since there's more than enough room to store the one-byte **char** value in the two-byte **int**. The sum of this addition is 30, another **int** value.

The next operation adds that partial sum to the **long** value of `l_val` (to make the expression easier to read, we'll show the sum from the previous addition):

```
30 + l_val
```

This addition triggers another promotion. The compiler promotes the **int** result of the first addition to a **long** value before adding it to `l_val`, which is **long**. Since the four-byte **long** type has more than enough room to store a two-byte **int**, this promotion is also harmless.

Now the partial sum equals 64030. The last addition from CONVERT.C

```
64030 + f_val
```

triggers another harmless conversion: the compiler converts the **long** result of the previous addition to a **float** value before adding it to `f_val`. Even though floating-point and integer values are stored in different internal formats, no data is lost when the **long** is converted to a **float**.

The result of these additions and conversions is the **float** value 64033.10, which is correct. So where does the mistake occur?

The problem arises when CONVERT.C assigns the final sum to the wrong type of variable. You'll recall that the line containing these operations begins with the assignment `result =`.

Earlier in the program, we declared the variable `result` as an **int**. The two-byte **int** variable created to store the result of these additions is too small to contain the four-byte **float** sum that was finally produced.

The assignment forces QuickC to demote the larger **float** value to the smaller **int** type. It's impossible to store such a large floating-point value in the two bytes of an **int**, so the final result is incorrect.

Figure 5.2 shows the progression of automatic type conversions that the CONVERT.C program produces.

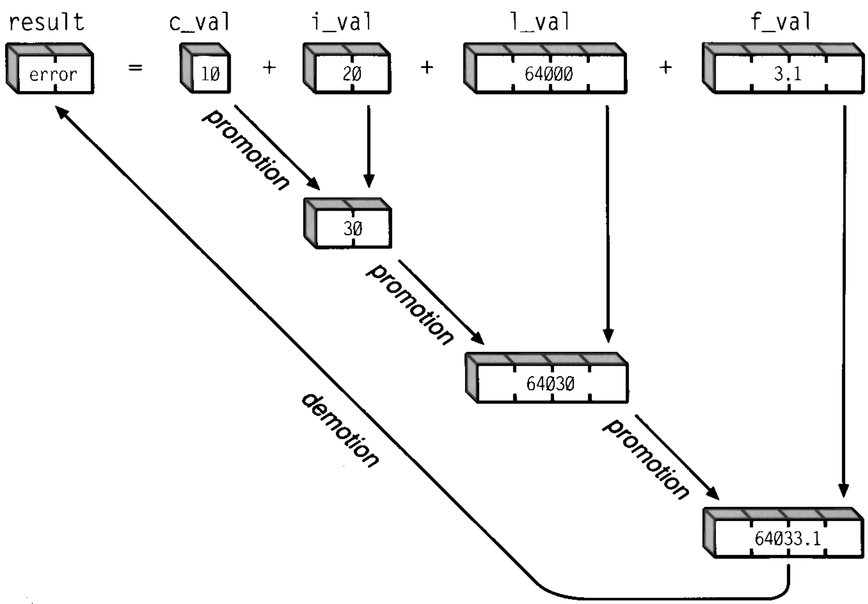


Figure 5.2 Automatic Type Conversions in CONVERT.C

We can fix the conversion error by declaring the variable `result` as a **float**, substituting

```
float result;
```

for the earlier declaration. We'll also need to change the format string in the **printf** function call to print a **float** value, as shown below:

```
printf( "%6.2f\n", result );
```

Now CONVERT.C prints the expected value of 64033.10.

Manual Type Conversions through Casting

A cast forces a value to a particular type.

The C language also allows you to force a type conversion that would not otherwise happen, a process known as "casting." Using casts, it is possible to convert a data item to any C data type.

Sometimes you must use a cast to make the program work properly. When calling the **malloc** library function, for instance, you should perform a cast on the value that the function returns. (Chapter 12, "Dynamic Memory Allocation," explains **malloc** and other memory-allocating functions.)

Casts can also make a program more readable. QuickC does most automatic type conversions silently. So if you write a tricky bit of code that relies on automatic conversions, you, or some other programmer, may not notice the conversions later. To make such code more readable—and easier to debug—you can add explicit type casts in places where silent conversions might go unnoticed.

To cast a value to a different type, place the desired type name in parentheses in front of the value. For instance, the statement

```
f_val = (float)any_val;
```

casts the value of the variable `any_val` to type **float** before assigning it to `f_val`. Here the type name in parentheses,

```
(float)
```

performs the cast. No matter what type `any_val` has, the cast converts that type to **float** before assigning it to `f_val`.

When you cast a variable, the cast affects the value the variable yields, but not the variable itself. Suppose that `any_val` is an **int** variable with the value 333. The above cast converts the value 333 to **float** format before assigning it to `f_val`. But `any_val` remains an **int** variable after the cast.

Remember, you can detect automatic type conversions by setting Warning Level 2 or higher in QuickC and watching for the following warning:

```
C4051: data conversion
```

You can then add explicit casts to eliminate the warning where the conversions are desirable. (See "Automatic Type Conversions" above.)

Register Variables

Register variables are stored in processor registers instead of addressable memory.

You can use the **register** keyword in variable declarations to request that a variable be stored in a processor register. Because processor registers can be accessed more quickly than addressable memory locations, this storage can make a program run faster. Programmers use **register** to speed access to heavily used variables, such as counter variables in loops.

The **register** specifier is much less important than it used to be, now that most C compilers, including QuickC, can perform optimizations (improvements) during compilation. If you compile with the Optimizations option turned on, QuickC automatically stores variables in registers when needed. So you probably won't need to use **register** except in special cases.

IMPORTANT *If you compile with Optimizations on, an explicit* ***register*** *declaration can override register storage that QuickC would do automatically. Declaring one variable with* ***register*** *might prevent QuickC from storing some other variable in a register. In the worst case, this can make a program run slower.*

You can use **register** only with short integer types (**char**, **int**, and **short int**). Other types—including aggregate types such as arrays—are too large to fit in a register.

Only two registers are available for variable storage at any given time. (They are DI and SI, for those who have programmed in assembly language.) If you request more registers than are available, QuickC stores the extra variables in addressable memory, as it does non-**register** variables.

The following declaration uses **register** to ask the compiler to store the **int** variable `val` in a processor register:

```
register int val;
```

You can ask the compiler to store more than one variable in a register. For instance, the statement

```
register int val, count;
```

declares `val` and `count` as **register** variables.

NOTE *Since registers are not addressable, you can't use the address-of (**&**) operator to get the address of a variable declared with **register**. This rule applies whether or not QuickC is actually able to store the variable in a register. Thus, if you need to access a variable through a pointer, don't declare that variable with **register**. See the section "Initializing a Pointer Variable" in Chapter 8, "Pointers."*

Renaming Existing Types with typedef

The **typedef** keyword creates a new name for an existing data type. This is a convenience feature that you can use to make programs more readable. For instance, the declaration

```
typedef int integer;
```

allows you to use `integer` as a synonym for **int**.

One more practical use of **typedef** is to substitute a short, descriptive name for an aggregate type. For instance, the QuickC Presentation Graphics library uses **typedef** to create descriptive names such as **windowtype** and **titletype** for structures used in that library.

NOTE *The **typedef** keyword doesn't create a new data type. It merely allows you to use a different name for a type that already exists.*

You can also use **typedef** to minimize portability problems. By using **typedef** declarations for data types that are machine dependent, you need only change the **typedef** declaration if you move the program to a different operating system.

The Enumeration Type

The "enumeration type" specifies a set of named integer constants, similar to the enumerated type in QuickPascal. In the C language, enumeration types are declared with the **enum** keyword.

Use enum to name a set of integer constants.

The **enum** type is useful mainly for improving a program's readability. With **enum**, you can use meaningful names for a set of constants whose purpose might not otherwise be apparent.

Suppose you're writing a calendar program in which the constant 0 represents Saturday, 1 represents Sunday, and so on. You might begin by declaring the enumeration type `day` in the following manner:

```
enum day
{
   saturday, sunday, monday, tuesday,
   wednesday, thursday, friday
};
```

Notice this declaration's similarity to a structure declaration. As with structures, the type declaration creates a template that you can use to declare variables of this type. (See the section "Declaring a Structure Type" in Chapter 4, "Data Types.")

Unless you specify otherwise, the first value in an enumeration type equals 0 and others are numbered sequentially. In the **enum** type shown above, `saturday` equals 0, `sunday` equals 1, and so forth.

The values in an enumeration type need not be sequential, however. If you want some other order, you can declare explicit values for each member of the type. The following declaration, for example, assigns the names `zero`, `freeze`, and `boil` to the constants 0, 32, and 220, respectively.

```
enum temps
{
   zero = 0,
   freeze = 32,
   boil = 220
};
```

After declaring an enumeration type, you can create a variable of that type and assign it a value from the type. The statement

```
enum day today = wednesday;
```

declares the variable `today`, assigning it the value `wednesday` from the `day` enumeration type.

After you assign its value, you can use the variable `today` as you would an **int** variable. Although the variable is considered to have the **enum** type, it is an ordinary **int** for all practical purposes.

Enumeration types aren't used very often, partly because you can achieve a similar effect using the **#define** directive. (Chapter 7, "Preprocessor Directives," explains **#define** in detail.) For example, the code

```
#define SATURDAY 0
#define SUNDAY 1
#define MONDAY 2
#define TUESDAY 3
#define WEDNESDAY 4
    .
    .
    .
int today = WEDNESDAY;
```

uses **#define** to create symbolic constants named `SATURDAY`, `SUNDAY`, `MONDAY`, `TUESDAY`, and `WEDNESDAY`, assigning them the values 0 through 4. The last line in the example creates the **int** variable `today` and assigns it the value of `WEDNESDAY`. The result is identical to the statement shown earlier:

```
enum day today = wednesday;
```

One advantage of using **enum** over **#define** directives is that it groups related names in one place and can be more compact than a long series of directives.

This concludes our main discussion of data types. The next chapter, "Operators," examines the C language's rich set of operators, which allow you to manipulate data in many different ways.

CHAPTER 6

Operators

Compared with other languages, C is very compact, using fewer than 50 keywords. One reason C can get by with so few reserved words is its abundance of powerful operators—well over 30.

Most C operators are easy to understand and remember. Even if you have never seen a C program, you probably understand that the statement

```
val = val * 5;
```

multiplies the variable `val` by 5 and assigns the result to `val`.

Because the printable ASCII character set has only so many unique symbols, C uses some ASCII symbols in more than one operator. For instance, the asterisk (*) performs either a multiplication or pointer operation, depending on context. Similarly, the ampersand (&) is part of three C operators. Depending on context, the ampersand can obtain an address or perform a logical or bitwise AND operation. Be careful not to confuse operators that look similar but do different jobs.

This chapter describes the C operators, beginning with those that are common to most languages, and then discussing those unique to C.

Introducing C's Operators

We'll start by discussing C operators that look and behave similarly to operators in other languages. These include the following groups:

- Arithmetic operators, which do operations such as addition and multiplication
- Relational operators, which compare two values and give a true or false result
- Assignment operators, which make one value equal to another

Arithmetic Operators

The C language's arithmetic operators closely resemble those in other languages. Table 6.1 lists C's arithmetic operators.

Table 6.1 Arithmetic Operators

Operator	Description
*	Multiplication
/	Division
%	Modulus
+	Addition
–	Subtraction

The "modulus operator" (%) may be unfamiliar. It divides a value and gives the remainder. For instance, the statement

```
remainder = 20 % 3;
```

assigns the value 2 to the variable `remainder` (20 divided by 3 equals 6, with a remainder of 2). If the division doesn't produce a remainder, the modulus operator yields the value 0.

Relational Operators

"Relational operators" evaluate the relationship between two expressions, giving a true result (the value 1) or a false result (the value 0). C has six relational operators, which are listed in Table 6.2.

Table 6.2 Relational Operators

Operator	Description
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal

The "equality operator" (==), shown above, tests whether two expressions are equal.

Don't confuse the equality operator with the assignment operator (=) discussed in the next section. The assignment operator sets one value equal to another, as we'll see shortly. (Chapter 10, "Programming Pitfalls," discusses this common programming error.)

The C language gives the value 1 for true and 0 for false but recognizes any nonzero value as true. The following code fragment demonstrates this difference:

```
printf( "C generates %d for true\n", 2 == 2 );
printf( "C generates %d for false\n", 2 == 4 );
if( -33 )
   printf( "C recognizes any nonzero value as true\n" );
```

The output from this code,

```
C generates 1 for true
C generates 0 for false
C recognizes any nonzero value as true
```

shows that the true expression (`2 == 2`) gives the value 1 and the false expression (`2 == 4`) gives the value 0. The last output line shows that C recognizes the nonzero value –33 as a true value.

Assignment Operators

The "assignment operator" (=) sets one value equal to another. The following statement assigns the value of `sample` to `val`:

```
val = sample;
```

You can combine an assignment with a bitwise or arithmetic operation.

In a convenient shorthand, C allows you to combine the assignment operator with any arithmetic or bitwise operator (see the "Arithmetic Operators" and "Bitwise Operators" sections). For example, the statement

```
val = val + sample;
```

can more conveniently be written

```
val += sample;
```

Both statements add `val` to `sample` and then assign the result to `val`.

Table 6.3 lists C's special assignment operators.

Table 6.3 Special Assignment Operators

Expression	Equivalent	Operation
`x *= y`	`x = x * y`	Multiplication
`x /= y`	`x = x / y`	Division
`x %= y`	`x = x % y`	Modulus
`x += y`	`x = x + y`	Addition
`x -= y`	`x = x - y`	Subtraction
`x <<= y`	`x = x << y`	Left shift
`x >>= y`	`x = x >> y`	Right shift
`x &= y`	`x = x & y`	AND
`x ^= y`	`x = x ^ y`	Exclusive OR
`x |= y`	`x = x | y`	Inclusive OR

Note that the equal sign always follows the other operator. In the following code,

```
val ^= sample;
val =^ sample;
```

the first statement is meaningful, but the second is a syntax error.

C's Unique Operators

The remaining sections in this chapter describe C operators that fall into two categories: those that are unique to C and those that look or behave differently in C than in other languages.

Increment and Decrement Operators

The C language's unique "increment" (++) and "decrement" (– –) operators are very useful. They increase or decrease an expression by a value of 1.

Table 6.4 Increment and Decrement Operators

Operator	Operation
++	Increment expression by 1
--	Decrement expression by 1

Thus, the two statements

```
val = val + 1;
val++;
```

are equivalent and so are these statements:

```
val = val   1;
val--;
```

You can use the ++ and -- operators before or after an expression.

The ++ and -- operators can precede or follow an expression. Placed before an expression, the operator changes the expression before the expression's value is used. In this case, the operator is said to be a "prefix" operator. Placed after an expression, the operator (known as a "postfix" operator) changes the value of the expression after the expression's value is used.

In the DECRMENT.C program, shown below, the decrement operator is used both as a prefix operator and a postfix operator.

```
/* DECRMENT.C: Demonstrate prefix and postfix operators. */

#include <stdio.h>
main()
{
   int val, sample = 3, proton = 3;
   val = sample--;
   printf( "val = %d  sample = %d\n", val, sample );
   val = --proton;
   printf( "val = %d  proton = %d\n", val, proton );
}
```

Here is the output from DECRMENT.C:

```
val = 3  sample = 2
val = 2  proton = 2
```

In the first use of the decrement operator, the statement

```
val = sample--;
```

assigns the value of `sample` (3) to the variable `val` and then decrements `sample` to the value 2. Contrast this with the statement

```
val = --proton;
```

which first decrements `proton` to the value 2 and then assigns that value to `val`.

Bitwise Operators

The "bitwise operators," listed in Table 6.5, manipulate bits in data of the integer type. These operators are often used in programs that must interact with hardware.

Table 6.5 Bitwise Operators

Operator	Description
~	Complement
<<	Left shift
>>	Right shift
&	AND
^	Exclusive OR
|	Inclusive OR

The ~ operator, known as the "one's complement," acts on only one value (rather than on two, as do most operators). This operator changes every 1 bit in its operand to a 0 bit and vice versa.

The << and >> operators, known as the "shift operators," shift the left operand by the value given in the right operand. These operators offer a fast, convenient way to multiply or divide integers by a power of 2.

The & operator, known as the "bitwise AND," sets a bit to 1 if either of the corresponding bits in its operands is 1, or to 0 if both corresponding bits are 0. It is often used to "mask," or turn off, one or more bits in a value.

The ^ operator, known as the "bitwise exclusive OR," sets a bit to 1 if the corresponding bits in its operands are different, or to 0 if they are the same.

The | operator, known as the "bitwise inclusive OR," sets a bit to 1 if either of the corresponding bits in its operands is 1, or to 0 if both corresponding bits are 0. It is often used to turn on bits in a value.

Each of the bitwise operators is used in the BITWISE.C program, shown below.

```
/* BITWISE.C: Demonstrate bitwise operators. */

#include <stdio.h>

main()
{
   printf( "255 & 15 = %d\n", 255 & 15 );
   printf( "255 | 15 = %d\n", 255 | 15 );
   printf( "255 ^ 15 = %d\n", 255 ^ 15 );
   printf( "2 << 2   = %d\n", 2 << 2 );
   printf( "16 >> 2  = %d\n", 16 >> 2 );
   printf( "~2       = %d\n", ~2 );
}
```

The output from BITWISE.C,

```
255 & 15 = 15
255 | 15 = 255
255 ^ 15 = 240
2 << 2   = 8
16 >> 2  = 4
~2       = -3
```

shows the results of the program's various bitwise operations.

The fourth and fifth output lines show you how to use shift operators to multiply and divide by powers of 2. The program multiplies 2 by 4 by shifting the value 2 twice to the left:

```
2 << 2   = 8
```

Similarly, the program divides 16 by 4 by shifting the value 16 twice to the right:

```
16 >> 2  = 4
```

Logical Operators

C has three logical operators—AND, OR, and NOT—that allow you to test more than one condition in a single expression. Table 6.6 lists C's logical operators.

Table 6.6 Logical Operators

Operator	Description
!	Logical NOT
&&	Logical AND
||	Logical OR

The logical OR (||) and AND (&&) operators are often used to combine logical tests within a conditional statement. For example, the **if** statement

```
if( val > 10 && sample < 10 )
   printf( "Oh joy!\n" );
```

prints `Oh joy!` if both conditions in the test expression are true (if `val` is greater than 10 and `sample` is less than 10). Here, the relational operators (> and <) have higher "precedence" than the logical AND operator (&&), so the compiler evaluates them first. We discuss operator precedence later in this chapter.

The logical NOT operator (!) reverses an expression's logical value. For instance, if the variable `val` has the value 8, the expression `(val == 8)` is true but the expression `!(val == 8)` is false.

The NOT.C program below shows a common use of this operator.

```
/* NOT.C: Demonstrate logical NOT operator. */

#include <stdio.h>

main()
{
  int val = 0;
  if( !val )
     printf( "val is zero" );
}
```

The expression `if( !val )` is equivalent to the expression `if( val == 0 )`. When used in this way, the logical NOT operator converts a 0 value to 1 and any nonzero value to 0.

***NOTE** Don't confuse the logical OR and AND operators with the bitwise OR and AND operators discussed in the previous section. The bitwise operators use the same ASCII symbols, but have only one character. For instance, logical AND is **&&**, whereas bitwise AND is **&**.*

Address Operators

The C language has two operators that work with memory addresses. Table 6.7 lists C's address operators.

Table 6.7 Address Operators

Operator	Operation
&	Yield address of the operand
*	Yield value contained at the operand's address

Both address operators are often used with pointers—variables that contain the addresses of other variables. Chapter 8, "Pointers," and Chapter 9, "Advanced Pointers," are devoted to explaining pointers, including the use of these two operators with them. Since you must understand pointers in order to understand these operators fully, we'll describe them briefly here and elaborate on their use in Chapter 8.

The "address-of operator" (&) yields a constant equal to the machine address of its operand. For instance, if the variable `val` contains the value 10, and its storage is located at address 1508, the expression `val` yields the value 10, while the expression `&val` yields the constant 1508.

Since the address-of operator yields a constant, you can't assign a value to an expression that uses it. The statement

```
&val = 20;
```

is illegal for the same reason that the statement

```
1508 = 20;
```

won't pass muster.

The "indirection operator" (*) yields the value contained in the address referenced by its operand. If you declare `ptr` as a pointer variable, the expression

```
*ptr
```

yields the contents of the address to which `ptr` points.

Conditional Operator

The "conditional operator" (**? :**) is made up of two symbols and requires three expressions. It is similar to an **if-else** construct. If the first expression evaluates as true, the first operand is assigned the value of the second operand. If the first expression is false, the first operand is assigned the value of the third operand.

The following statement gives the absolute value of the variable `val`. The variable is assigned its original value if it is nonnegative or is negated if its original value is negative:

```
val = (val >= 0 ) ? val : -val;
```

The statement is equivalent to the following **if-else** construct:

```
if( val >= 0 )
   val = val;
else
   val = -val;
```

The sizeof Operator

The "**sizeof** operator" yields the number of bytes contained in its operand, which can be either a general data type or a specific variable. If you apply **sizeof** to a type name in parentheses, as in the expression

```
sizeof( int )
```

the operator yields the size of that data type in bytes. This example yields the value 2, indicating that an **int** contains two bytes on DOS machines. You can use this feature to determine the sizes of types that are implementation dependent when transporting a program from one machine to another.

If you place **sizeof** in front of a variable name, the operator returns the number of bytes in the variable. For instance, if you create the string

```
char my_string[] = "Hello";
```

the expression

```
sizeof my_string
```

yields the value 6, showing that the string contains 5 printing characters and a null character.

Comma Operator

Preceding chapters have shown various ways to use the comma (,) in C programming. For instance, commas can separate multiple function arguments or variable declarations. In such cases the comma is not an operator in the formal sense but merely punctuation, like the semicolon that ends a statement.

The comma is used as punctuation and as an operator in C.

In C, the comma can also perform as an operator. The commas that separate multiple expressions determine the order in which the expressions are evaluated, and the type and value of the result that is returned. The comma operator causes expressions to be evaluated from left to right. The value and type of the result are the value and type of the rightmost operand.

For example, the statement

```
val = sample, sample = temp;
```

first assigns the value of `sample` to `val`, then assigns the value of `temp` to `sample`.

The comma operator often appears in **for** statements, where it can separate multiple initializing expressions or multiple modifying expressions. The FORLOOP1.C program from Chapter 3, "Flow Control," demonstrates both uses. Here is the **for** statement from that program:

```
for( a = 256, b = 1; b < 512; a = a / 2, b = b * 2 )
   printf( "a = %d \tb = %d\n", a, b );
```

The statement initializes two variables (`a` and `b`) and contains two modifying expressions (`a = a / 2` and `b = b * 2`). Chapter 3 explains the FORLOOP1.C program in detail.

Base Operator

The base operator (:>) associates a base expression with a based pointer. Based-object support is a highly advanced feature included in QuickC 2.5 for compatibility with Microsoft C version 6.0; please refer to your C 6.0 documentation for information about based objects.

Operator Precedence

Like all languages, C has precedence rules that control the order for evaluating the elements in expressions containing more than one operator. If you're familiar with precedence rules in other languages, you won't find any surprises in C. Table 6.8 shows the "pecking order" established for C's operators.

Three general rules control the order of evaluation:

1. When two operators have unequal precedence, the operator with higher precedence is evaluated first.
2. Operators with equal precedence are evaluated from left to right.
3. You can change the normal order of precedence by enclosing an expression in parentheses. The enclosed expression is then evaluated first. (If parentheses are nested, inner parentheses have higher precedence than outer ones.)

We'll demonstrate operator precedence with a simple example. Since the multiplication operator (*) has higher precedence than the addition operator (+), the statement

```
val = 2 + 3 * 4
```

assigns to `val` the value of 14 (or 2 + 12) rather than 20 (or 5 * 4). Since parentheses have higher precedence than any operator, they can change the normal precedence order. If you enclose the addition operation in parentheses, as follows

```
val = (2 + 3) * 4
```

the addition is done first. Now the statement assigns to `val` the value 20 (or 5 * 4).

Table 6.8 lists the C operators and their precedence values. The lines in the table separate precedence levels. The highest precedence level is at the top of the table.

Table 6.8 C Operators

Symbol	Name or Meaning	Associativity
( )	Function call	Left to right
[ ]	Array element	
.	Structure or union member	
->	Pointer to structure member	
--	Decrement	Right to left
++	Increment	
:>	Base operator	Left to right
!	Logical NOT	Right to left
~	One's complement	
–	Unary minus	
+	Unary plus	

Table 6.8 C Operators *(continued)*

Symbol	Name or Meaning	Associativity
&	Address	
*	Indirection	
sizeof	Size in bytes	
(type)	Type cast [for example, (**float**) i]	
*	Multiply	Left to right
/	Divide	
%	Modulus (remainder)	
+	Add	Left to right
−	Subtract	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal	Left to right
!=	Not equal	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Conditional	Right to left
=	Assignment	Right to left
*=, /=, %=, +=, −=, <<=, >>=, &=, ^=, |=	Compound assignment	
,	Comma	Left to right

CHAPTER 7

Preprocessor Directives

This chapter describes preprocessor directives—commands that control the QuickC compiler. It explains how to insert the contents of one source file into another file, how to do text substitutions throughout a file, and how to compile different parts of a file in different situations.

A “preprocessor directive” is a command to the QuickC compiler. Although they appear in the same source file as executable statements, preprocessor directives aren’t statements in the formal sense. Unlike executable statements, they are not translated into machine code. Instead, they tell the compiler itself to take some action while it translates your source program. For instance, an **#include** directive tells QuickC to insert another file into the source file.

The term “preprocessor” refers to the time when these commands are carried out. Like most language compilers, QuickC translates your source program in several phases, the first of which is called the “preprocessor phase.” QuickC first “preprocesses” all the directives in your source program, then processes the program’s executable statements.

All preprocessor directives begin with a number sign (#), which must be the first nonblank character in the line on which it appears. Since directives aren’t statements, they don’t end with semicolons. You can’t put other statements or directives on the same line with a preprocessor directive, except for a comment, which must appear to the right of the directive.

Because the compiler reads your source file sequentially, line by line, the position of directives is important. A preprocessor directive only affects statements that follow it in the source file.

The #include Directive

The #include directive inserts another file in the source file.

The **#include** directive inserts the contents of another file into your source file. The inserted file is called an include file or header file.

When the compiler encounters an **#include**, it searches for the file named in the directive. This directive makes QuickC look for the standard include file STDIO.H:

```
#include <stdio.h>
```

If the designated file is found, the compiler inserts its contents at the spot where the **#include** directive appears. Figure 7.1 illustrates a program SAMPLE.C that includes the file STDIO.H.

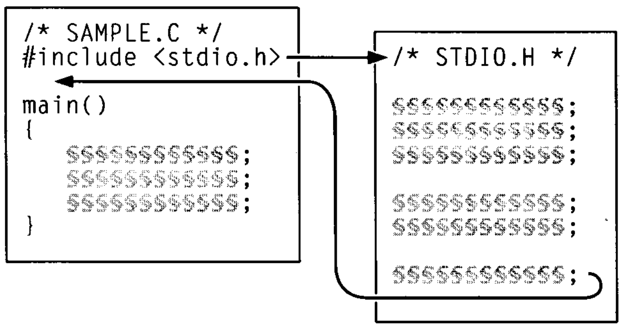


Figure 7.1 The #include Directive Inserts a File

When QuickC compiles the SAMPLE.C program shown in Figure 7.1, it inserts the contents of file STDIO.H into SAMPLE.C at the spot marked by the **#include** directive.

Most include files contain commonly used declarations and definitions. Standard include files, supplied with QuickC, contain declarations and definitions for QuickC library routines. You can also write include files of your own.

Standard include files end with the .H file extension (STDIO.H is an example). You can use any extension for include files you create, but most programmers stick with the .H extension.

NOTE *In some languages, it's common to put executable statements, as well as declarations and definitions, in include files. This practice is legal but not recommended in QuickC. Microsoft debugging tools such as the Microsoft CodeView® debugger may not recognize executable statements in include files.*

The **#include** directive doesn't support wild cards, so you can't insert a group of related files with a single directive. Each **#include** directive inserts only one file.

Include files can be nested. For instance, the source program SAMPLE.C might include a file named INOUT.H. The INOUT.H file, in turn, might contain a second **#include** directive that includes a file named KEYBOARD.H. Figure 7.2 illustrates this process.

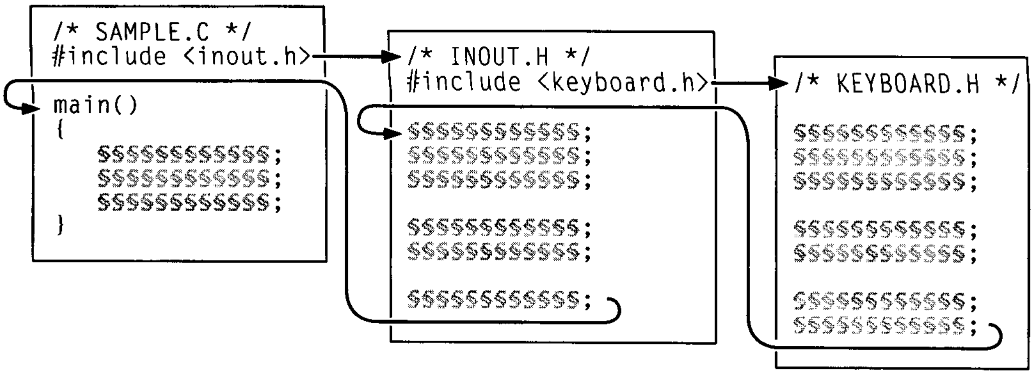


Figure 7.2 Nested Include Files

Although it's rarely necessary to nest include files more than two or three levels, nesting may continue up to 10 levels.

Specifying Include Files

There are two ways to tell QuickC where to search for an include file. You may have noticed that the **#include** directive shown earlier encloses the file name STDIO.H in angle brackets (<>). If you enclose the file name in angle brackets, as in the directive

```
#include <stdio.h>
```

the compiler searches the "standard directories" for the file.

In QuickC, the standard directories are one or more directories that you define by a DOS environment variable named INCLUDE. An advantage of specifying the

standard directories is that QuickC can automatically search more than one directory.

Alternatively, you can enclose the file name in double quotes, in the following manner,

```
#include "myfile.h"
```

to cause QuickC to start searching in the directory that contains the current source file. If the target file isn't in that directory, the compiler searches the standard directories.

NOTE *You can specify additional directories on the DOS command line when you invoke QuickC with the QCL command. See Chapter 1, "Creating Executable Programs," in the Microsoft QuickC Tool Kit.*

The #define and #undef Directives

The **#define** directive performs a text substitution in the source file. This directive has two main uses: simple text replacement and creation of function-like macros. It is also used with the **#undef** directive to control conditional compilation, as we'll discuss later.

Simple Text Replacement

The #define directive works like the search and replace function of a word processor.

At the simplest level, the **#define** directive works much like the "search and replace" function of a word processor, replacing one piece of text in the source file with another piece of text. The **#define** directive is commonly used to create a symbolic constant—a meaningful name for a "magic number" whose meaning might not otherwise be apparent. This improves the program's readability.

For instance, in the VOLUME.C program in Chapter 1, "Anatomy of a C Program," the directive

```
#define PI 3.14
```

defines a symbolic constant named `PI`. The directive causes QuickC to replace every occurrence of the text `PI` in the VOLUME.C source program with the text `3.14`. For example, when the compiler encounters the program line

```
result = 4 * PI * result;
```

it expands the line to read

```
result = 4 * 3.14 * result;
```

Besides making your program more readable, symbolic constants can streamline its maintenance. For instance, say you later decide to use 3.14159265 rather than 3.14 in VOLUME.C. All you need to change is one **#define** directive at the beginning of the program.

The replacement text can be longer than the 3.14159265 we used above. A replacement text can't be longer than 512 bytes in QuickC, but you'll rarely, if ever, have to worry about this limit.

Function-Like Macros

A function-like macro accepts arguments, like a function.

Some languages use the term "macro" when referring to replacement text. In C, a macro can do more than simply replace text. It can also accept arguments in much the same way that a function does. In this case the replacement text is called a "function-like macro."

A well-designed macro can be every bit as useful as a function. In fact, some C library routines are implemented as macros rather than C functions.

The MACRO.C program below has a macro that works like the **abs** library function, returning the absolute value of any integer. The macro uses the conditional operator (**? :**), which we explained in Chapter 6.

```
/* MACRO.C: Demonstrate macros. */

#include <stdio.h>
#define ABS(value)  ( (value) >= 0 ? (value) : -(value) )

main()
{
   int val = -20;
   printf( "result = %d\n", ABS(val) );
}
```

The `ABS` macro behaves much like a function. You can "call" it by name, passing it an argument you want to process. The macro is defined in the following program line:

```
#define ABS(value)  ( (value) >= 0 ? (value) : -(value) )
```

The parameter `value` appears four times in the macro—once in the macro name `ABS` and three times in the replacement text that follows the name.

Always enclose macro parameters in parentheses.

To avoid unwanted side effects, you should always enclose macro parameters in parentheses. If the parameter passed to the `ABS` macro is an expression containing operators, the lack of parentheses could cause operator-precedence errors. See the section "Omitting Parentheses from Macro Arguments" in Chapter 10, "Programming Pitfalls."

Function-like macros, like other **#define** directives, are expanded during the preprocessor phase of compilation, before QuickC translates any executable statements. When QuickC encounters the line

```
printf( "result= %d\n", ABS(val) );
```

it expands it to read:

```
printf( "result= %d\n", ( (val) >= 0 ? (val) : -(val) ) );
```

Macros can improve readability.

A macro can improve a program's readability by describing the nature of an operation while hiding its complex details. Most people find the first of the two statements above easier to understand than the expanded version.

Macros are faster than functions but can make a program bigger.

Function-like macros are fast, too. Since a macro creates in-line code, it doesn't have the overhead associated with a normal function call. On the other hand, each use of a macro inserts the same code in your program, whereas a function definition occurs only once. So while macros can be faster than functions, they can also bloat the size of the executable file.

The #undef Directive

The "**#undef** directive" is related to **#define**. As the name suggests, **#undef** removes ("undefines") a name that was created with **#define**. For instance, if you create the symbolic constant `PI` with the **#define** directive,

```
#define PI  3.14
```

you can then remove the name `PI` with the following **#undef** directive:

```
#undef PI
```

You can use **#define** and **#undef** to create a name that has meaning in only part of a source program. The next two sections explain why you might want to do this.

Conditional Directives

Conditional directives are useful for making different versions of a program.

Conditional directives can make QuickC skip part of a source file. They are used primarily to create different versions of a program. While developing a program, for instance, you might want to include debugging code at some times but not others. Or, if you plan to move a program to some other machine, you can compile machine-specific sections of code only for a certain machine.

The C-language conditional directives are listed below.

#if	**#endif**
#else	**#ifdef**
#elif	**#ifndef**

NOTE *The **#ifdef** and **#ifndef** directives are obsolete under the ANSI C standard; see "The defined Operator" below.*

The #if directive works like the if statement.

The **#if** and **#endif** directives work like an **if** statement, allowing you to compile a block of source code if a given condition is true. The **#if** directive is followed by a constant expression, which the compiler tests at compile time. If the expression is false, the compiler skips every line between the **#if** and the next **#endif**.

The example below calls the `display` function only if the name `DEBUG` was previously defined as 1 (with **#define**).

```
#if DEBUG == 1
   display( debuginfo );
#endif
```

Here, the "conditional block" is a single line (the `display` function call). A conditional block can contain any number of valid C program lines, including preprocessor directives as well as executable statements.

The test expression for a conditional directive can be almost any expression that evaluates to a constant, with a few minor exceptions (the expression can't use the **sizeof** operator, type casts, or the **float** and **enum** types).

The #else directive works like the else keyword.

The **#else** and **#elif** directives work like the **else** keyword and can perform more complex conditional tests. For example, you could use code like that in the following example to build different versions of a program for various IBM PC computers, including different files for each computer.

```
#if XT == 1
   #include "XT.H"
#elif AT == 1
   #include "AT.H"
#else
   #include "PS2.H"
#endif
```

The code includes the file XT.H if the name `XT` is defined as 1 and it includes the file AT.H if the name `AT` is defined as 1. If both `XT` and `AT` are undefined, the third conditional block executes, causing QuickC to include the file PS2.H.

You can nest conditional directives in the same way as you would conditional C language statements.

The defined Operator

The defined operator tests whether a name has been defined.

The test expression of an **#if** or **#elif** directive can use the **defined** operator to test whether a name has been defined. You can use this feature, along with **#define** and **#undef**, to turn various parts of a program on and off, compiling different parts under different conditions.

The **defined** operator is true if its argument has been defined and false otherwise. A name is considered defined if it has been created with **#define** (and not later removed with **#undef**).

The DEFINED.C program below prints `Hi` because the name `DEBUG` is defined when the compiler encounters the **#if defined** directive.

```
/* DEFINED.C: Demonstrate defined operator. */

#define DEBUG 12345

main()
{
   #if defined( DEBUG )
      printf( "Hi\n" );
   #endif
}
```

The **defined** operator tests only whether a name is defined, not whether it has a certain value. Thus, the DEFINED.C program will print `Hi` no matter what value is assigned `DEBUG`. You could substitute the directive

```
#define DEBUG 0
```

to define `DEBUG` as 0, or the directive

```
#define DEBUG
```

to define `DEBUG` as having no value at all. Both directives define the name `DEBUG`, so the program would print `Hi` in both cases.

You can use the logical NOT operator (**!**) to reverse the logic of an **#if defined** directive. (Logical operators are explained in Chapter 6.) The code

```
#if !defined( DEBUG )
   printf( "Hi\n");
#endif
```

prints `Hi` if `DEBUG` is not currently defined.

A plain **#if** directive treats undefined names a little differently than does an **#if defined** directive. If a name is not currently defined, the **#if** directive treats the name as having the value 0.

In the following code, the **#if** directive explicitly tests whether `DEBUG` equals 0.

```
#undef DEBUG
#if DEBUG == 0
   printf( "Hi\n" );
#endif
```

The result is the same as that of the previous example.

NOTE *The **defined** operator is new under the ANSI C standard. You may see older programs that use the older directives **#ifdef** and **#ifndef** for the same purpose. These directives are obsolete, but QuickC version 2.5 supports them for the sake of compatibility. The **#ifdef** directive is followed by a name (not in parentheses) and works the same as **#if** with **defined**. If the given name has been defined, **#ifdef** is true. The **#ifndef** directive is the opposite of **#ifdef**. It is true if the given name is not currently defined.*

Pragmas

Pragmas are implementation-specific compiler commands.

Although portability is a hallmark of C, the language's creators recognized that every C compiler will need to support some features unique to its host machine. The "**#pragma** directive" offers a way for each C compiler to offer machine-specific features while retaining overall compatibility with the C language. Since pragmas are machine-specific by definition, they can be—and usually are—different for every C compiler.

Pragmas have the same general syntax as preprocessor directives. The pragma must begin with a number sign (#) and it can't share a line with other directives or statements except a comment, which must appear to the right of the pragma.

QuickC supports four pragmas: **check_stack**, **check_pointer**, **message**, and **pack**. Each of these pragmas is described in online help.

Some pragmas take arguments, which come after the **#pragma** keyword. In the following code, the **message** pragma displays different messages during compilation depending on the outcome of an **#if** test:

```
#if XT == 1
   #pragma message( "Building XT version" )
#elif AT == 1
   #pragma message( "Building AT version" )
#endif
```

The message displayed by the **message** pragma is visible only if you compile from the DOS command line with the QCL command.

CHAPTER

8 Pointers

The next two chapters explain pointers—a large and important topic in C. This chapter explains fundamental techniques: how to use pointers with various data types and pass them to functions. In Chapter 9, "Advanced Pointers," we'll explore more advanced pointer techniques, such as multiple indirection.

If you have never used pointers before, you may want to read this chapter now and then turn to Chapter 9 after you have had some practice using pointers in your own programs.

Don't panic! There's a lot of new information in these two chapters. Don't be discouraged if you don't grasp it all on a first reading. The idea behind a pointer is simple, but some advanced pointer techniques are not so easy to follow at first.

Using Pointers in C

Almost every real-world C program uses pointers in some way or another. Much of the usefulness of pointers stems from the fact that in C all function arguments are passed by value. Because a function only receives local copies of such arguments, it can't change the original values that the arguments represent. Pointers make this possible.

Here are some common uses of pointers:

- Manipulating strings
- Passing command-line arguments to a program at run time
- Returning more than one value from a function
- Accessing variables that wouldn't otherwise be visible to a function

- Manipulating an array by moving pointers to its elements instead of using array subscripting
- Accessing the address of a memory area that your program allocates at run time
- Passing the address of one function to another function

Pointers to Simple Variables

A pointer variable contains the address of a data object.

Although pointers have many different uses, it takes only a few words to say what a pointer is. A "pointer" is a variable that contains the address of some other data object—usually a variable. Because a pointer contains the other variable's address, it is said to "point to" that variable.

This section uses the program POINTER.C to demonstrate the basic mechanics of pointers—how to declare and initialize a pointer and use it to access a simple variable:

```
/* POINTER.C: Demonstrate pointer basics. */

#include <stdio.h>

main()
{
   int val = 25;
   int *ptr;
   ptr = &val;
   printf( " val = %d\n", val );
   printf( "*ptr = %d\n\n", *ptr );
   printf( "&val = %u\n", &val );
   printf( " ptr = %d\n", ptr );
}
```

Here is the output from POINTER.C:

```
 val = 25
*ptr = 25

&val = 5308
 ptr = 5308
```

(The third and fourth output lines show addresses. These may differ when you run POINTER.C depending on factors such as available memory.)

POINTER.C creates a pointer variable named `ptr` and makes `ptr` point to an **int** variable named `val`. Then it prints the two values to show that `ptr` can access the value stored in `val`. The program goes on to print the address where `val` is stored and the address contained in `ptr`, to show they are the same.

Declaring a Pointer Variable

Like any variable, a pointer variable must be declared before it is used, and its value can change in the course of a program. A pointer variable can have any legal variable name. Here is the pointer declaration from POINTER.C:

```
int *ptr;
```

This declaration states the program has a pointer variable named `ptr` that can point to a data object of the **int** type.

Notice the similarity to a simple variable declaration. As in other cases, the declaration gives a type (**int**) and name (`ptr`) for the variable.

Use the indirection operator (*) to declare a pointer variable.

The indirection operator (*) in front of the name `ptr` shows this variable is a pointer. This operator has two different uses in C. In declarations, such as the one above, it simply means "this is a pointer." In other contexts, as we'll elaborate throughout this chapter, it means indirection—using the data object that a pointer points to.

A pointer declaration shows what type of data object a pointer references.

A pointer doesn't have a type in the same sense as other variables. When you declare a simple variable, the type specifier shows what type of value the variable stores. When you declare a pointer variable, the type specifier shows what type of data object the pointer *points to*.

Thus, in POINTER.C the declaration of the variable `val` indicates `val` stores a value of the type **int**,

```
int val = 25;
```

while the declaration of the variable `ptr` indicates it *points to* a data object of the type **int**:

```
int *ptr;
```

To declare pointers to other types of variables, you can use whatever type specifier is appropriate. These statements, for instance, declare pointers to **char** and **float** variables:

```
char *c_ptr, *ch;
float *f_pointer;
```

Note that if you declare more than one pointer variable in the same line, each name must be preceded by the indirection operator. The first line in the previous example declares two pointer variables: `c_ptr` and `ch`. Each pointer can point to an object of the **char** type. If you omit the second indirection operator from the first line,

```
char *c_ptr, ch;
```

the line declares a pointer variable named `c_ptr` and an ordinary **char** variable named `ch`.

A pointer declared with type void can point to any type of data object.

In most cases a pointer points to a particular type of object, such as an **int**. You can also declare a pointer with type **void**, which allows it to point to any type of object.

One use of **void** pointers is to write a general-purpose function, such as a sort, that can operate on data of more than one type. Each time you use a **void** pointer, you must perform an explicit type cast to show what type of object it points to on that occasion.

Figure 8.1 shows the relationship between `val` and `ptr` in POINTER.C, immediately after `ptr` has been declared. The figure shows that the variable `val` is stored at memory location 5308, as in the output shown above. Again, the actual address may differ when you run POINTER.C.

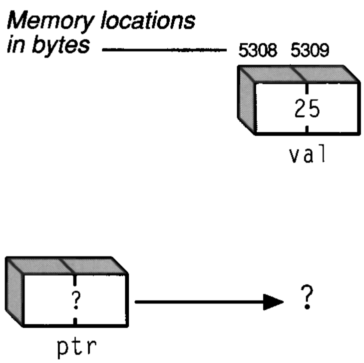


Figure 8.1 Before Initialization of ptr

Figure 8.1 uses question marks to show that the contents of `ptr` are undefined at this stage in the program. Like any other variable that has been declared but not initialized, the contents of `ptr` are unknown. You must take special care not to use pointers that have not been initialized, since an uninitialized pointer might point anywhere in memory—including sensitive operating-system addresses.

WARNING *Because a pointer can potentially access any memory address, using an uninitialized pointer can have drastic consequences.*

How Pointers Are Stored

Figure 8.1 also shows that while a pointer is a special kind of variable, it is not a mysterious entity floating in limbo. A pointer is a true variable whose contents are stored at a specific memory address.

In POINTER.C we don't care precisely where the pointer's contents are stored—the compiler handles that detail for us, as it does so many others. So Figure 8.1 does not include the address of the storage for `ptr`. It does show, however, that the pointer is stored in two bytes, the same amount of memory needed to store an **int** value.

NOTE *The actual amount of memory needed to store a pointer variable depends on the current "memory model." In the small memory model—the default for QuickC version 2.5—a pointer is stored in two bytes. In some larger memory models, a pointer is stored in four bytes. For purposes of discussion, this chapter and the following chapter assume the small memory model. Appendix B, "Working with QuickC Memory Models," in the Microsoft QuickC Tool Kit discusses memory models.*

Initializing a Pointer Variable

The next step in the POINTER.C program is to initialize the pointer variable `ptr`, making it point to some meaningful address in memory:

```
ptr = &val;
```

The "address-of operator" (&) gives the address of the name it precedes. So in plain English the above statement says, "assign the address of `val` to `ptr`."

After its initialization, the variable `ptr` points to `val` in the sense that it contains the address where `val` is stored.

The output from POINTER.C shows that `ptr` contains the address of `val`. First it prints the address of `val` using the address-of operator to directly obtain the variable's address,

```
&val = 5308
```

then it prints the contents of `ptr`:

```
ptr = 5308
```

The two values are identical. Figure 8.2 shows the relationship of `val` and `ptr` at this stage in the POINTER.C program.

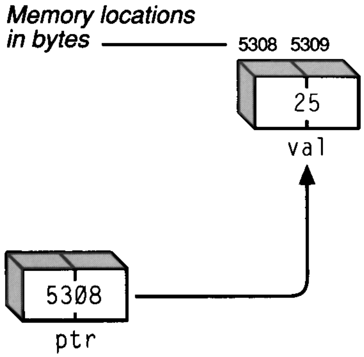


Figure 8.2 After Initialization of ptr

Initialization is especially important for pointers because, as noted earlier, they have the potential to point anywhere in memory. If you forget to initialize it, or make it point to the wrong place, a pointer can wreak havoc with your program or even the operating system itself.

The target of a pointer must be present in memory at run time.

The pointer in POINTER.C points to a simple **int** variable. As a general rule, pointers can point to any data object that is present in memory at run time. This category mainly includes objects for which the program allocates (sets aside) memory. Memory can be allocated implicitly, by defining a variable or function, or explicitly, by calling a memory-allocating library function such as **malloc**.

A pointer can't point to program elements such as expressions or **register** variables, which aren't present in addressable memory.

POINTER.C initializes the pointer `ptr` by assigning it an address constant (the address of `val`, obtained with the address-of operator). You can also assign the value of one pointer to another, as shown here:

```
ptr = ptr1;
```

If `ptr` and `ptr1` are both pointers, this statement assigns the address contained in `ptr1` to `ptr`.

Using a Pointer Variable

Once `ptr` points to `val`, we have two ways to access the **int** value stored in `val`. The usual way is direct, using the name of `val`:

```
printf( " val = %d\n", val );
```

The second way to access `val` is indirect, using the pointer variable `ptr` and the indirection operator:

```
printf( "*ptr = %d\n\n", *ptr );
```

Both of the preceding statements print the value of `val`, confirming that you can access the contents of `val` indirectly as well as directly. Once `ptr` points to `val` you can use `*ptr` anywhere that you would use `val`.

The indirection operator can obtain the value to which a pointer points.

Using the indirection operator to access the contents of `val` is the second use of this operator (the first is in declaring pointer variables, as explained earlier). When the asterisk appears in front of the name `ptr`, the expression states that you want to use the *value the pointer points to*, not the value of `ptr` itself.

The second **printf** statement in POINTER.C uses the expression `*ptr` to access the value stored in `val`.

This use of a pointer is analogous to the **PEEK** function in QuickBASIC. You can just as easily use `ptr` to change the data in `val`, an operation that somewhat resembles a QuickBASIC **POKE** statement.

For instance, if you add the following statements to the end of POINTER.C,

```
*ptr = 3301;
printf( "%d\n", val );
```

the program prints `3301`.

Figure 8.3 shows the relationship between `ptr` and `val` after executing the previous two statements.

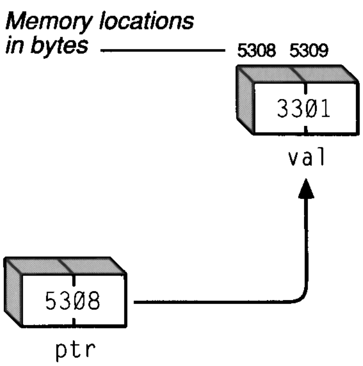


Figure 8.3 Changing Contents of val with ptr

As Figure 8.3 shows, the value stored in `val` has changed from 25 to 3301. The contents of `val` were changed indirectly, through the pointer `ptr`.

Summary of Pointer Basics

In the preceding sections, you have seen how to do these operations:

- Declare a pointer to a simple variable
- Initialize a pointer, making it point to a variable
- Use a pointer to get the value of a variable
- Use a pointer to change the contents of a variable

It's important for you to be comfortable with these basic ideas before reading about the more advanced uses of pointers. If you're not sure you understand these concepts, you may want to experiment with the POINTER.C program to reinforce what you know. For instance, you might add some new variables of different types and create new pointers to access them.

Pointers to Arrays

Pointers and arrays are closely related in C—a major theme we'll elaborate throughout the rest of this chapter and Chapter 9, "Advanced Pointers." This section explains one of the simpler ways to use pointers with arrays.

A pointer to an array, or "array pointer," combines two powerful language features—the pointer's ability to provide indirect access and the convenience of accessing array elements through numerical subscripts.

An array pointer can point to any element in a given array.

A pointer to an array is not much different than a pointer to a simple variable. In both cases, the pointer can point only to a single object at any given time. An array pointer, however, can reference any individual element within an array (but just one at a time).

The program PARRAY.C shows how to access the elements of an **int** array through a pointer:

```
/* PARRAY.C: Demonstrate pointer to array. */

#include <stdio.h>

int i_array[] = { 25, 300, 2, 12 };

main()
{
   int *ptr;
   int count;
   ptr = &i_array[0];
```

```
    for( count = 0; count < 4 ; count++ )    {
        printf( "i_array[%d] = %d\n", count, *ptr );
        ptr++;
    }
}
```

Here is the output from PARRAY.C:

```
i_array[0] = 25
i_array[1] = 300
i_array[2] = 2
i_array[3] = 12
```

The PARRAY.C program creates a four-element **int** array named `i_array`. Then it declares a pointer named `ptr` and uses `ptr` in a **for** loop to access each of the elements in `i_array`.

Notice the similarity between PARRAY.C and the previous example (POINTER.C). The pointer is declared in the same way:

```
int *ptr;
```

As noted before, this declaration states that `ptr` can point to any object of the **int** type, which includes an element in an **int** array as well as a simple **int**. The initialization of `ptr` looks similar, too:

```
ptr = &i_array[0];
```

This statement assigns `ptr` the address of the first element of `i_array`, which is `i_array[0]`. (There's a more compact way to initialize this pointer, but we'll defer that discussion for a moment.) Figure 8.4 shows the relationship between `ptr` and `i_array` immediately after `ptr` is initialized.

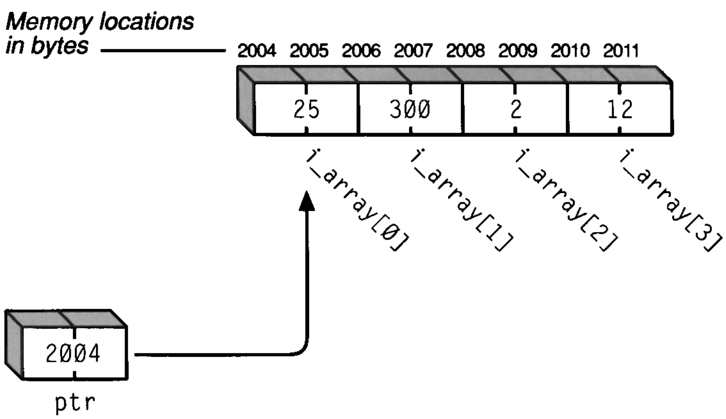


Figure 8.4 Pointer to an Integer Array

Arrays and Pointer Arithmetic

Once a pointer points to an array, it can access any of the array's elements. By adding or subtracting from the pointer's value (using "pointer arithmetic") you can access any element in the array, just as you can access it with array subscripts.

So in PARRAY.C, just as in POINTER.C, we can use `*ptr` to access the **int** value that `ptr` references. The only difference is now `ptr` points to an array element instead of a simple variable.

When the **for** loop in PARRAY.C executes the first time, `ptr` points to the first element of `i_array`, which is `i_array[0]`. The second statement in the loop body,

```
ptr++;
```

increments the pointer. Now `ptr` points to the next element in `i_array`, which is `i_array[1]`. Figure 8.5 shows the relationship of `ptr` and `i_array` after the first iteration of the **for** loop in PARRAY.C.

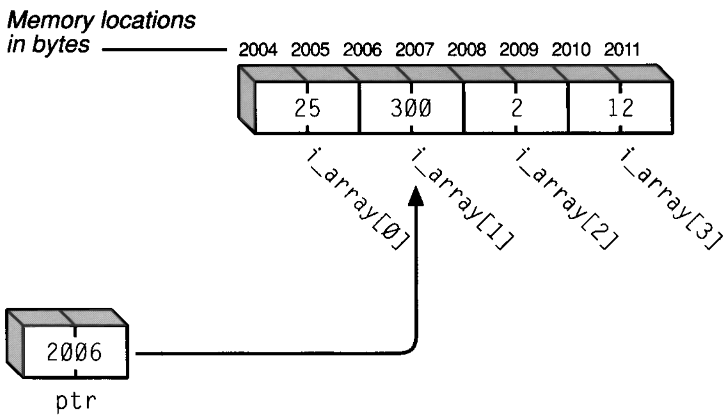


Figure 8.5 After Incrementing ptr

Figures 8.4 and 8.5 illustrate another important fact about pointers. Pointer arithmetic is automatically scaled to the size of the object that a pointer references. As explained above, incrementing `ptr` with the statement

```
ptr++;
```

Pointer arithmetic is scaled to the size of elements in an array.

moves the pointer forward to the next *element* in `i_array`. Since each element of an **int** array contains two bytes, this operation actually adds 2 to the address stored in `ptr`, but you don't have to worry about that detail. The compiler knows the size of the elements in the array and adjusts the pointer accordingly.

Incrementing a pointer adds 1 if it points to a **char** array, 4 if it points to a **float** array, and so on.

You can also decrement an array pointer. If `ptr` points to `i_array[2]`, this statement moves the pointer back one element, to `i_array[1]`:

```
ptr--;
```

Although the previous expressions increment and decrement `ptr` by 1, you can add or subtract any integer value from a pointer. For instance, the following statement moves `ptr` forward three elements in `i_array`:

```
ptr += 3;
```

Be careful not to overrun the bounds of an array when accessing its elements with a pointer. As noted in Chapter 4, "Data Types," the C language doesn't check array subscripts. This rule applies equally when you access an array with a pointer, which can potentially reference any address in memory.

WARNING *The C language does not check array pointer references. If you increment or decrement a pointer past the limits of an array, you can corrupt other parts of your program or cause other unexpected results.*

It's your job to make sure an increment or decrement doesn't move a pointer outside the memory where an array is stored. For instance, if you decrement `ptr` when it points to `i_array[0]`, it will point to whatever happens to be stored in the **int**-size memory area *below* the element `i_array[0]`.

Most pointer arithmetic occurs in connection with arrays, where a numerical index has obvious utility. It's not illegal to do pointer arithmetic on nonarray pointers, but such operations normally serve no purpose. For instance, if you increment a pointer to a simple variable, the pointer no longer points to the variable and becomes useless.

Comparing Pointers

The special nature of a pointer variable—the fact that it contains an address—precludes most operations that are legal for other variables. There's no such thing as a fractional memory address, for example. So it wouldn't make sense to divide a pointer, or add a floating-point number to it. The most common pointer operations are assignment, incrementing, and decrementing, as described earlier. You can also compare one pointer to another.

If a program allocates memory for a stack, for instance, you might create two pointers that point to different parts of the stack. One pointer can show where the stack begins and the other where it ends. To see how much of the stack is in use, you can subtract the pointers. (A "stack" is a memory area used for temporary storage.)

You can compare pointer variables with relational operators or by subtraction.

Pointer comparisons can be done with relational operators (such as <) or by subtracting one pointer from another. Of course, pointer comparisons are meaningful only for pointers that point to the same data object or related objects of the same type.

PARRAY.C Revisited

Before leaving the PARRAY.C program, we should note that most C programmers would write it more compactly (PARRAY1.C):

```
/* PARRAY1.C: Compact version of PARRAY.C. */

#include <stdio.h>

int i_array[] = { 25, 300, 2, 12 };

main()
{
   int count;
   int *ptr = i_array;
   for( count = 0; count < 4 ; count++ )
      printf( "i_array[%d] = %d\n", count, *ptr++ );
}
```

You can declare and initialize a pointer variable in one statement.

The PARRAY1.C program uses several shorthand techniques you can expect to see in C programs. Like other variables, pointers can be initialized at the same time they are declared. The following statement in PARRAY1.C performs both operations:

```
int *ptr = i_array;
```

The statement above is equivalent to these statements:

```
int *ptr;
ptr = i_array;
```

You may have noticed another difference in the way `ptr` is initialized. The PARRAY1.C program omits the address-of operator and array subscript that PARRAY.C used to signify the address of the first element of `i_array`. Instead of

```
&i_array[0]
```

the program uses

```
i_array
```

An array name is a pointer.

In fact, the two expressions are equivalent. In the C language, the name of an array is actually a pointer. Any array name that doesn't have a subscript is interpreted as a pointer to the base address of the array. (The "base address" is the address of the array's first element.) We'll explore this equivalence further in the following sections and in Chapter 9, "Advanced Pointers."

Finally, PARRAY1.C uses the expression `*ptr++` to perform two jobs: accessing the value `ptr` points to and incrementing `ptr`. Note the order in which the two operators in this expression take effect. The indirection operator takes effect first, accessing the value of the array element that `ptr` currently points to. Then the increment operator (++) adds 1 to `ptr`, making it point to the next element in `i_array`.

Pointers and Strings

String pointers are handled like other array pointers.

Because a string is an array of characters, pointers to strings are handled much like other array pointers. The program PSTRING.C is similar to the examples that demonstrated array pointers (PARRAY.C and PARRAY1.C). It uses a pointer to access a **char** array:

```
/* PSTRING.C: Demonstrate pointer to a string. */

#include <stdio.h>

main()
{
   int count;
   char name[] = "john";
   char *ptr = name;
   for( count = 0; count < 4; count++ )
   {
      printf( "name[%d]: %c\n", count, *ptr++ );
   }
}
```

The PSTRING.C program steps through the `name` array, printing each character in turn:

```
name[0]: j
name[1]: o
name[2]: h
name[3]: n
```

The notable difference between PARRAY.C and PSTRING.C is that PSTRING.C has a **char** array instead of an **int** array. Again, incrementing an

array pointer moves the pointer to the next array element. So in PSTRING.C each iteration of the **for** loop moves the pointer to the next **char** in the string.

The first time through the loop, `ptr` points to `name[0]`. The second time it points to `name[1]`, and so on.

As mentioned in Chapter 4, "Data Types," one difference between strings and noncharacter arrays is that strings end with a null character. The string in PSTRING.C actually contains five characters: four letters and a null character. We can exploit this fact to simplify the program, as we do below in PSTRING.C.

```
/* PSTRING1.C: Look for null at string's end.  */

#include <stdio.h>

main()
{
   char name[] = "john";
   char *ptr = name;
   while( *ptr )
      printf( "*ptr = %c\n", *ptr++ );
}
```

Here is the output from PSTRING1.C:

```
*ptr = j
*ptr = o
*ptr = h
*ptr = n
```

Like PSTRING.C, the PSTRING1.C program steps through the array one character at a time. However, it replaces the **for** loop with a simpler **while** loop. The test expression in the **while** loop,

```
while( *ptr )
```

is evaluated as true until `ptr` points to the null character that terminates the string. It's a more compact way of writing this expression:

```
while( *ptr != 0 )
```

Any operation done with array subscripts can also be done with pointer notation.

This is an ideal time to elaborate on the relationship between arrays and pointers. Any operation you can do with conventional array notation (subscripts) can also be done with pointers. This is possible because an array name, as we noted earlier, is itself a pointer.

To illustrate, the PSTRING2.C program uses only array notation:

```
/* PSTRING2.C: Demonstrate strings and array notation. */

#include <stdio.h>
#include <string.h>
```

```
main()
{
   int count;
   char name[] = "john";
   for( count = 0; count < strlen( name ); count++ )
      printf( "name[%d]: %c\n", count, name[count] );
}
```

PSTRING2.C gives the same output as PSTRING.C. In this program, the expression

```
name[count]
```

uses `count` as in an index to the `name` array.

PSTRING3.C is the same program written with pointer notation:

```
/* PSTRING3.C: Strings and pointer notation.  */

#include <stdio.h>
#include <string.h>

main()
{
   int count;
   char name[] = "john";
   for( count = 0; count < strlen( name ); count++ )
      printf( "*(name+%d) = %c\n", count,*(name+count) );
}
```

Here is the output from PSTRING3.C:

```
*(name+0) = j
*(name+1) = o
*(name+2) = h
*(name+3) = n
```

Notice how PSTRING3.C replaces the expression

```
name[count]
```

with the expression:

```
*(name+count)
```

Both expressions use the variable `count` as an offset from the base address of the array. The parentheses in the second expression are important. They are necessary because the indirection operator takes effect before the addition operator. If you omit the parentheses, as in

```
*name+count
```

the expression has the same effect as

```
(*name)+count
```

which adds the value of `count` to the object `name` references.

In summary, the examples in this section show three alternative ways to access a character inside a string. In the **printf** statements in the examples, these expressions are equivalent:

```
*ptr
```

```
name[count]
```

```
*(name+count)
```

Many C programmers prefer pointer notation to array notation because pointers are faster for some operations. In other cases—including the one above—the choice is entirely one of taste. There's more to say about the relationship between pointers and arrays. We'll return to this topic later in this chapter and in Chapter 9, "Advanced Pointers."

Passing Pointers to Functions

A function that receives pointers can access variables that are local to other functions.

One of the most common uses of pointers is to pass them as arguments to functions. Functions that receive variables as parameters get local copies of those variables, not the originals. In contrast, functions that receive pointers to variables gain access to the original variables associated with the pointers. This allows the functions to

- Return more than one value
- Read and change values in variables—including arrays and structures—that otherwise aren't visible to the function

The first item listed above relates to the **return** statement. As we noted in Chapter 2, "Functions," a function can return only one value through **return**. However, it's not difficult to imagine a useful function—a sort, for instance—that would return more than one value. Pointers offer an elegant solution.

The second item involves visibility. Most variables in C programs are local to the functions where they are defined, and a function normally can't access local variables in other functions. There are times, however, when you want a function to have access to a local variable defined elsewhere in the program. By passing the function a pointer to the local variable, you can give it access to the variable itself.

The PFUNC.C program illustrates both ideas. It has a function that returns more than one value and uses pointers to alter variables that aren't visible within the function:

```
/* PFUNC.C: Pass pointers to a function. */

#include <stdio.h>

void swap( int *ptrl, int *ptr2 );

main()
{
   int first = 1, second = 3;
   int *ptr = &second;
   printf( "first: %d  second: %d\n", first, *ptr );
   swap( &first, ptr );
   printf( "first: %d  second: %d\n", first, *ptr );
}

void swap( int *ptrl, int *ptr2 )
{
   int temp;
   temp = *ptrl;
   *ptrl = *ptr2;
   *ptr2 = temp;
}
```

Here is the output from PFUNC.C:

```
first: 1  second: 3
first: 3  second: 1
```

Pointers can eliminate the need for external variables.

The PFUNC.C program swaps the values of two **int** variables named `first` and `second`, using a function named `swap`. Since the exchange involves two values, the `swap` function can't use **return** to communicate its results. Moreover, the variables `first` and `second` are defined only in the **main** function, and as good C programmers, we want to exchange their values without making them externally visible.

The prototype for the `swap` function shows that `swap` expects to receive two pointers to **int** variables:

```
void swap( int *ptrl, int *ptr2 );
```

Notice the use of **void** in the prototype and function definition. The **void** specifier shows that the `swap` function doesn't return any value through a **return** statement. Instead, `swap` returns its results indirectly, through the action of pointers.

The variables we want to exchange are defined only in **main**:

```
int first = 1, second = 3;
```

No other function in the program can access these variables directly by using the variable names `first` and `second`. We must pass these variables as arguments; but since the C language passes arguments by value, we need to pass pointers to the variables.

The **main** function calls `swap` with the following statement:

```
swap( &first, ptr );
```

This statement shows two different ways to pass a pointer to a function. The first argument in the function call,

```
&first
```

passes the address of first as a constant, using the address-of operator. The second argument,

```
ptr
```

passes the address of `second` with a pointer variable. Earlier in PFUNC.C we declared `ptr` as a pointer to an **int** and assigned it the address of `second`:

```
int *ptr = &second;
```

Both arguments pass the same kind of data—the address of a local variable—to the function. We'll return to this idea after we see how the rest of PFUNC.C works.

When the `swap` function executes, it creates two **int** pointers named `ptr1` and `ptr2` and assigns the passed addresses to them:

```
void swap( int *ptr1, int *ptr2 )
```

Since there's a one-to-one correspondence between arguments and parameters, the pointer `ptr1` receives the address of `first` and `ptr2` receives the address of `second`. The `swap` function exchanges the values of `first` and `second`, using the two pointers and a temporary **int** variable named `temp`:

```
int temp;
temp = *ptr1;
*ptr1 = *ptr2;
*ptr2 = temp;
```

Within the `swap` function, PFUNC.C uses the indirection operator to access the values that `ptr1` and `ptr2` reference. The expression `*ptr1` accesses the value stored in `first`. Likewise, the expression `*ptr2` accesses the value stored in `second`.

Through the addresses contained in the pointers, the `swap` function can indirectly access variables that are local to the **main** function.

Passing Address Constants Versus Passing Pointer Variables

Now that you know how the `swap` function works, we can elaborate on the two methods that PFUNC.C uses to pass the address of `first` and `second` to `swap`.

When you pass a pointer to a function, the function actually receives an address.

Earlier, we said the `swap` function expects to receive two pointers as parameters. While it's common to say pointers in this context, it would be more accurate to say the function expects addresses, since that's what it actually receives.

To work correctly, `swap` only needs the addresses of two variables. Once it has the addresses, it assigns them to its own local pointers and proceeds to do its work—modifying the original variables at long distance, as it were. The `swap` function doesn't care whether you pass the addresses as constants or pointer variables, since it receives the same kind of value in either case. The address is all the function needs to change the value of a variable defined elsewhere.

The first argument in the function call to `swap` shows a straightforward way to pass an address. Inside the **main** function of PFUNC.C, the expression `&first` equals the address of `first`. When you pass this argument to `swap`, the function clearly receives an address.

The second argument is an address, too. Since **main** assigns the address of `second` to the pointer variable `ptr`, the expression `ptr` equals the address of `second`. When you pass this argument to `swap`, the function also receives an address. (Remember, the value contained in a pointer variable is an address.)

Some beginning programmers get confused by functions that expect to receive pointers, thinking they must always pass pointer *variables* to such functions. As PFUNC.C shows, if the function expects an address you can simply pass the address as a constant, using the address-of operator.

***NOTE** When a function expects to receive an address as a parameter, you can pass either an address constant or a pointer variable, whichever is more suitable.*

Why, then, would you ever go to the trouble of passing a pointer variable to this kind of function? In a real program, the function that calls `swap` might well use pointers to process `first` and `second` for some other purpose. In such a case you might prefer to use pointers in the function call, too.

Arrays of Pointers

Pointers, like other variables, can be stored in arrays. This feature allows you to create a variety of useful data structures.

In an array of pointers, each array element is a pointer variable.

If you find an array of pointers hard to picture, begin with the idea that an array is a group of variables of the same type. An "array of pointers" is also a group of variables, but instead of simple variables, it contains a group of pointer variables.

Each element in an array of pointers, then, is a pointer that contains an address. Like other array elements, each element can be accessed with a numerical subscript.

Pointer arrays are often used to speed up sorts. The QCSORT.C program shows the basic idea behind such a sort:

```
/* QCSORT.C: Demonstrate sorting array of pointers. */

#include <stdio.h>
#define SIZE 4

void sort( int size, double *p[] );
void show( int size, double *p[], double dd[] );

main()
{
   int x;
   double d[] = { 3.333, 1.111, 2.222, 4.444 };
   double *d_ptr[SIZE];
   for( x = 0; x < SIZE; x++ )
      d_ptr[x] = &d[x];
   show( SIZE, d_ptr, d );
   sort( SIZE, d_ptr );
   show( SIZE, d_ptr, d );
}

void sort( int size, double *p[] )
{
   int x, x1;
   double *temp;
   for( x = 0; x < size - 1; x++ )
      for( x1 = x + 1; x1 < size; x1++ )
      {
         if( *p[x] > *p[x1] )
         {
            temp = p[x1];
            p[x1] = p[x];
            p[x] = temp;
         }
      }
}
```

```
void show( int size, double *p[], double dd[] )
{
   int x;
   printf( "-----------------------" );
   printf( "-----------------------\n" );
   for( x = 0; x < size; x++ )
   {
      printf( "*d_ptr[%d] = %1.3f   ", x, *p[x]);
      printf( "d_ptr[%d] = %u ", x, p[x]);
      printf( "  d[%d] = %1.3f\n", x, dd[x] );
   }
}
```

Here is the output from QCSORT.C:

```
----------------------------------------------
*d_ptr[0] = 3.333  d_ptr[0] = 66   d[0] = 3.333
*d_ptr[1] = 1.111  d_ptr[1] = 74   d[1] = 1.111
*d_ptr[2] = 2.222  d_ptr[2] = 82   d[2] = 2.222
*d_ptr[3] = 4.444  d_ptr[3] = 90   d[3] = 4.444
----------------------------------------------
*d_ptr[0] = 1.111  d_ptr[0] = 74   d[0] = 3.333
*d_ptr[1] = 2.222  d_ptr[1] = 82   d[1] = 1.111
*d_ptr[2] = 3.333  d_ptr[2] = 66   d[2] = 2.222
*d_ptr[3] = 4.444  d_ptr[3] = 90   d[3] = 4.444
```

Since the purpose of QCSORT.C is to demonstrate pointers, not sorting methods, it uses a simple bubble sort. This method isn't efficient but has the advantage of being short and easy to follow.

The QCSORT.C program creates a **double** array named `d` and an array of pointers named `d_ptr`. Each array has four elements. To illustrate the sort, the elements of `d` are initialized out of order.

The goal of QCSORT.C is to display a sorted list of the values in `d`. You could do this by sorting the elements of `d` itself, but that solution is not efficient. Every **double** value contains eight bytes, and sorting a large number of **double** values requires that you move a lot of memory.

Instead of moving the **double** values themselves, QCSORT.C creates an array of pointers that point to the elements of the `d` array, then sorts the pointers. This saves time because a pointer is stored in only two bytes. Figure 8.6 shows the relationship between the `d` and `d_ptr` arrays immediately after both are initialized.

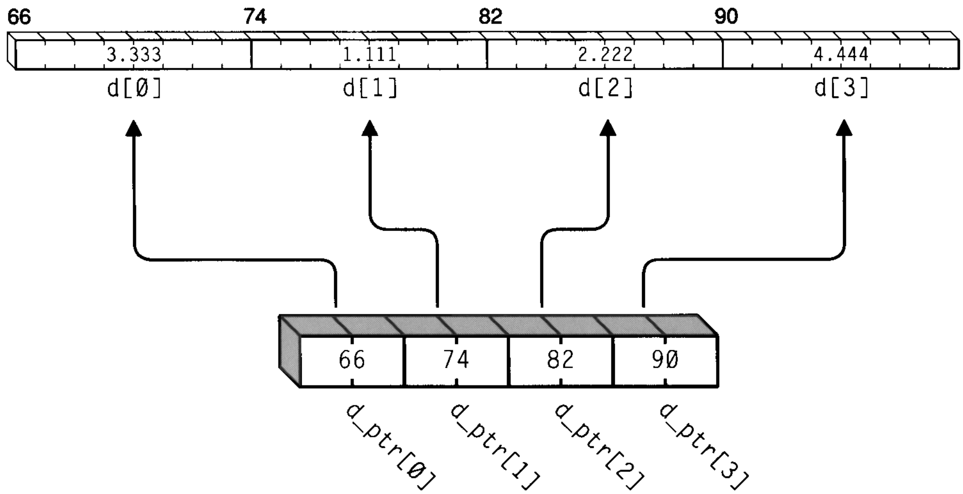


Figure 8.6 Before Sorting Array of Pointers

At the stage shown in Figure 8.6, the pointers in the `d_ptr` array have been initialized to point to the **double** elements in the `d` array. (The array element `d_ptr[0]` points to `d[0]`, `d_ptr[1]` points to `d[1]`, and so on.) The function `show` displays three sets of data:

- The value each pointer references
- The address assigned to each pointer
- The value of each element in the `d` array

After calling the `show` function, QCSORT.C calls the `sort` function, which sorts the pointers in `d_ptr`.

The declaration of `sort` contains something new. In the declaration

```
void sort( int size, double *p[] );
```

the expression `*p[]` shows that the `sort` function expects to receive a *pointer to an array of pointers*. When the program calls `sort`, it passes the size of the array to be sorted (first argument) and a pointer to the array of pointers (second argument):

```
sort( SIZE, d_ptr );
```

Now the `sort` function has all the information it needs to sort the pointers in the `d_ptr` array, making each pointer point to the correct element in the `d` array.

After the sort is complete, QCSORT.C calls `show` again to display the results of the sort. Now that the pointers have been sorted, they can be used to display a sorted list of **double** values. Figure 8.7 shows the relationship between the `d` and `d_ptr` arrays after the sort is complete.

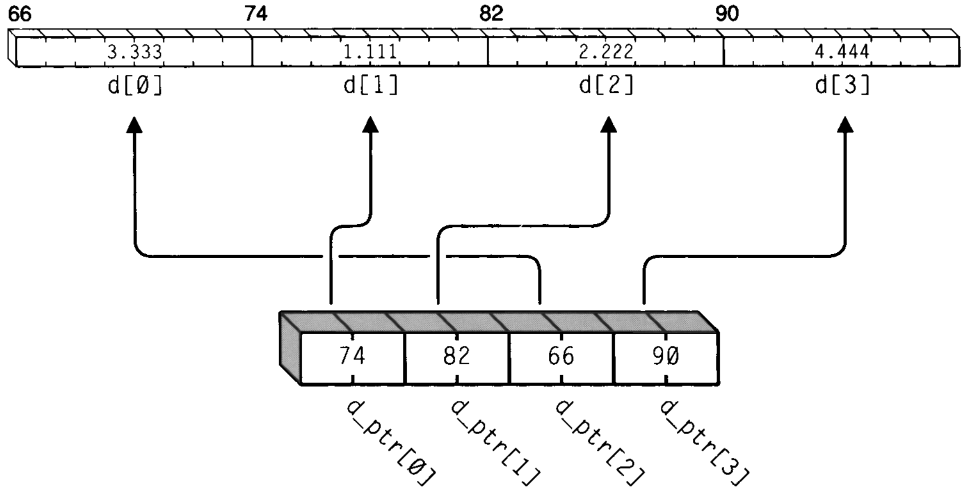


Figure 8.7 After Sorting Array of Pointers

Of course, the array in QCSORT.C is so small that the time savings from using pointers is negligible. In a real program, however, which might sort thousands of values instead of four, the difference between moving eight bytes and two bytes can be dramatic. The advantage of sorting pointers is even greater when sorting large data objects such as strings or structures.

The elements in a pointer array can point to any type of data.

The QCSORT.C example section uses a fairly simple array of pointers. But you can use such arrays to create quite complex data structures. The basic form of the array is always the same—it is a group of pointer variables, each pointer accessible through a subscript—but the pointers in an array can point to any kind of data object. You can have an array of pointers to structures, an array of pointers to strings, and so on. The only difference is in what the pointers reference.

Don't confuse an array of pointers with a pointer to an array. A pointer to an array (or "array pointer") is a single pointer variable that points to an array element. The single pointer can access any element of the array, but only one pointer is involved.

In contrast, an array of pointers is a group of related pointer variables stored in an array. Each element in the array is a pointer, and you can access individual pointers with the array name and subscript. Each pointer in the array points, in turn, to some other object.

A Pause for Reflection

If this is your first exposure to pointers, you may want to reflect on what you have learned before reading the next chapter. This chapter has explained the basic uses of pointers, and you can write a great many useful programs using only these techniques. If you're not comfortable with all these ideas, you may want to experiment with them before reading more about pointers.

The next chapter, "Advanced Pointers," examines further uses of pointers, including multiple indirection and pointers to structures.

CHAPTER

Advanced Pointers

9

The preceding chapter, "Pointers," explained the basics of using pointers—how to declare and initialize pointer variables and use them to access basic data types. This chapter explores more advanced pointer techniques, including multiple indirection, pointers to structures, and pointers to functions.

Pointers to Pointers

In Chapter 8, "Pointers," we stated a pointer can point to any kind of variable. Since a pointer is a variable, you can make it the target of another pointer, creating a pointer to a pointer. This concept is useful in itself and is also important for understanding the equivalence of array notation and pointer notation, which is explained in the next section.

The program PTRPTR.C demonstrates a pointer to a pointer in simple terms:

```
/* PTRPTR.C: Demonstrate a pointer to a pointer. */

#include <stdio.h>

main()
{
   int val = 501;
   int *ptr = &val;
   int **ptr_ptr = &ptr;
   printf( "val = %d\n", **ptr_ptr );
}
```

Here is the output from PTRPTR.C:

```
val = 501
```

The first two statements in PTRPTR.C should look familiar by now. They create an **int** variable named `val` and an **int** pointer named `ptr`. The third line, however, requires some explanation:

```
int **ptr_ptr = &ptr;
```

This statement uses double indirection to create a variable named `ptr_ptr`, which is a pointer to a pointer. This pointer is assigned the address of the first pointer, `ptr`. The pointer `ptr` references `val`, and the pointer `ptr_ptr` references `ptr`. Figure 9.1 illustrates the relationship between `ptr` and `ptr_ptr`.

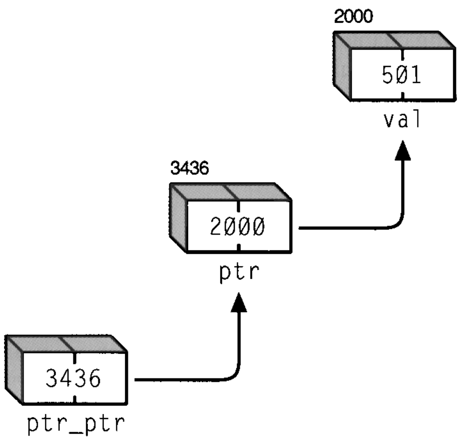


Figure 9.1 A Pointer to a Pointer

Once we have initialized both pointers, we can use `ptr_ptr` to access `val`:

```
**ptr_ptr
```

*The double indirection operator (**) is used with a pointer to a pointer.*

The double indirection operator (**) in front of `ptr_ptr` tells two things about `ptr_ptr`: that `ptr_ptr` is itself a pointer and it points to a second pointer. Both asterisks are needed to access the contents of `val`. If you use only one, as in

```
*ptr_ptr
```

then `ptr_ptr` accesses the contents of `ptr`, which is the address of `val`. This statement, for instance, prints the address stored in `ptr`:

```
printf( "ptr = %u", *ptr_ptr );
```

Using pointers to pointers is known as "multiple indirection." One pointer points to a second pointer, which in turn accesses a third data object. In theory, there's no limit to how far you can take multiple indirection. You can create pointers to pointers, pointers to pointers to pointers, and so on. However, there's rarely any practical reason to carry indirection beyond two levels (a pointer to a pointer).

Equivalence of Array and Pointer Notation

In previous sections we noted, more or less in passing, two important facts about arrays and pointers:

1. An array name is actually a pointer.
2. Array notation (subscripts) and pointer notation are interchangeable.

These ideas are significant enough to warrant an explicit demonstration. Let's rewrite the QCSORT.C program using pointer notation:

```
/* QCSORT1.C: Demonstrate sort with pointer notation. */

#include <stdio.h>
#define SIZE 4

void sort( int size, double **p );
void show( int size, double **p, double dd[] );

main()
{
   int x;
   double d[] = { 3.333, 1.111, 2.222, 4.444 };
   double *d_ptr[SIZE];
   for( x = 0; x < SIZE; x++ )
      d_ptr[x] = &d[x];
   show( SIZE, d_ptr, d );
   sort( SIZE, d_ptr );
   show( SIZE, d_ptr, d );
}
```

```
void sort( int size, double **p )
{
   int x, x1;
   double *temp;
   for( x = Ø; x < size - 1; x++ )
      for( x1 = x + 1; x1 < size; x1++ )
      {
         if( **(p+x) > **(p+x1) )
         {
            temp = *(p+x1);
            *(p+x1) = *(p+x);
            *(p+x) = temp;
         }
      }
}

void show( int size, double **p, double dd[] )
{
   int x;
   printf( "------------------------" );
   printf( "-----------------------\n" );
   for( x = Ø; x < size; x++ )
   {
      printf( "*d_ptr[%d] = %1.3f   ", x, **(p+x) );
      printf( "d_ptr[%d] = %u ", x, *(p+x) );
      printf( "  d[%d] = %1.3f\n", x, dd[x] );
   }
}
```

The QCSORT1.C program works like its predecessor, QCSORT.C. (It sorts an array of pointers that point to elements in an **int** array.) The only difference is QCSORT1.C uses pointer notation instead of array notation.

Let's look at how the change affects the `sort` function, beginning with its prototype. In the previous program, QCSORT.C, the prototype

```
void sort( int size, double *p[] );
```

uses array notation to show we'll pass the name of an array of pointers to `sort`. Since an array name is a pointer, we can rewrite the prototype using pointer notation, as in QCSORT1.C:

```
void sort( int size, double **p );
```

The `sort` function definition is rewritten in the same way. Here is the definition of `sort` in the original program (QCSORT.C):

```
void sort( int size, double *p[] )
{
   int x, x1;
   double *temp;
   for( x = 0; x < size - 1; x++ )
      for( x1 = x + 1; x1 < size; x1++ )
      {
         if( *p[x] > *p[x1] )
         {
         temp = p[x1];
         p[x1] = p[x];
         p[x] = temp;
         }
      }
}
```

The same function using pointers looks like this in QCSORT1.C:

```
void sort( int size, double **p )
{
   int x, x1;
   double *temp;
   for( x = 0; x < size - 1; x++ )
      for( x1 = x + 1; x1 < size; x1++ )
      {
         if( **(p+x) > **(p+x1) )
         {
            temp = *(p+x1);
            *(p+x1) = *(p+x);
            *(p+x) = temp;
         }
      }
}
```

Within the `sort` function, the variable `p` is a pointer to a pointer. When we use a single asterisk, as in,

```
*(p+x1)
```

we access the contents of the `x1` pointer, which is an address. When we place a double asterisk in front of an address value, as in,

```
**(p+x)
```

we access the contents of this address.

Using pointer notation in place of array notation, QCSORT1.C achieves the same result as QCSORT.C. In many cases—including this one—it doesn't really matter which notation you use. If you're still more comfortable with array notation, you may prefer to use it sometimes. Since many C programs use pointers to manipulate arrays, however, it's worth taking the time to learn pointer notation, too.

Getting Command-Line Arguments

Command-line arguments are passed to programs through argv, an array of pointers.

Arrays of pointers have one very common use—accessing command-line arguments. When a C program begins execution, DOS passes two arguments to it. The first argument, normally called `argc`, is an **int** variable that indicates the number of command-line arguments. The second, normally called `argv`, is a pointer to an array of strings. Each string in the array contains one of the command-line arguments.

Even if you don't plan to use `argc` and `argv` in your programs, you can expect to see them often in other C programs, so it's useful to know how they're used. The ARGV.C program uses `argc` and `argv`.

```
/* ARGV.C: Demonstrate accessing command-line arguments. */

#include <stdio.h>

void show_args( char *argument );

int main( int argc, char *argv[] )
{
   int count;
   for( count=0; count < argc; count++ )
      show_args( argv[count] );
   return 0;
}

void show_args( char *argument )
{
   printf( "%s\n", argument );
}
```

To make ARGV.C produce output, you must give it some command-line arguments. (If you run ARGV.C in the QuickC environment, select Run/Debug from the Options menu and type the command-line arguments at the Command Line prompt.) The program prints each argument on the screen.

If you use this command line, for instance,

```
argv harpo chico groucho zeppo
```

then ARGV.C gives this output:

```
C:\SOURCES\ARGV.EXE
harpo
chico
groucho
zeppo
```

The first argument may have surprised you. In DOS versions 3.0 and higher, the first string in the `argv` array (`argv[0]`) contains the drive specification and full pathname to the program that is executing. The drive and path you see will depend on how your system is configured. In the example the ARGV.EXE program is located in the SOURCES directory of drive C.

Thus, the value of `argc` actually is one greater than the number of command-line arguments, and the first argument typed on the command line is the second string in the array (`argv[1]`). If you type the arguments shown above, the value of `argc` is 5 and `argv[1]` contains the argument `harpo`.

Null Pointers

We can use the ARGV.C program to illustrate another handy property of pointers: null pointers. Consider this modification (ARGV1.C):

```
/* ARGV1.C: Demonstrate null pointers. */

#include <stdio.h>

void show_args( char *argument );

int main( int argc, char **argv )
{
   while( *argv )
      show_args( *(argv++) );
   return 0;
}

void show_args( char *argument )
{
   printf( "%s\n", argument );
}
```

The ARGV1.C program gives the same output as the previous program but it uses a **while** loop instead of a **for** loop. The test expression in this loop,

```
while( *argv )
```

is equivalent to this test expression:

```
while( *argv != 0 )
```

The loop in ARGV1.C continues until it finds a "null pointer," a pointer that contains 0. In this case, the null pointer means we have reached the end of the array: no more strings are available.

Null pointers can be used to show success or failure and as markers in a series.

Many C library functions use null pointers to signal the success or failure of an operation that returns a pointer. For instance, the library function **malloc** normally returns a pointer to the beginning address of the memory area it allocates. If no memory is available, **malloc** returns a null pointer to show the operation failed. Similarly, the **fopen** function usually returns a pointer to a **FILE** structure, but returns a null pointer when it fails.

Null pointers can also be used to mark the end of a list of pointers, such as the `argv` array or a linked list.

Pointers to Structures

A structure pointer can access any member of a structure.

A pointer to a structure, or "structure pointer," is conceptually similar to an array pointer. Just as an array pointer can point to any element in an array, a structure pointer can reference any member in a structure. The major difference is one of notation.

In case you're not yet an expert on structure notation, let's review it very briefly. First recall that each element in an array has the same type, so you refer to individual array elements with subscripts:

```
i_array[3]
```

Because members of a structure can have different types, you can't use numerical subscripts to refer to them based on their order. Instead, each structure member has a symbolic name. You refer to a member with a structure name and member name, separating the two names with the member-of operator (**.**):

```
jones.name
```

The notation for structure pointers follows the same pattern, with only two differences. You must

1. Replace the structure name with the name of the pointer
2. Replace the member-of operator with a two-character operator called the "pointer-member" operator (**–>**)

The pointer-member operator is formed by a dash and a right-angle bracket. The following name uses the pointer-member operator:

```
jones_ptr->name
```

Here `jones_ptr` is the name of a pointer to a structure, and `name` is a member of the structure that `jones_ptr` points to.

The EMPLOY1.C program is a revision of the EMPLOYEE.C program that demonstrates structures in Chapter 4, "Data Types." This program illustrates how to manipulate a structure through a pointer:

```
/* EMPLOY1.C: Demonstrate structure pointers. */

#include <stdio.h>

struct employee
{
   char name[10];
   int months;
   float wage;
};

void display( struct employee *e_ptr  );

main()
{
   struct employee jones =
   {
      "Jones, J",
      77,
      13.68
   };

   display( &jones );
}

void display( struct employee *e_ptr )
{
   printf( "Name: %s\n", e_ptr->name );
   printf( "Months of service: %d\n", e_ptr->months );
   printf( "Hourly wage: %6.2f\n", e_ptr->wage );
}
```

Structure pointers allow functions to access structures that are local to other functions.

The EMPLOY1.C program gives the same output as the earlier version. But instead of passing the entire structure to the `display` function, this program passes a structure pointer. This method conserves memory, since the `display` function doesn't create a local copy of the structure. It also allows `display` to change members in the original structure, which is local to the **main** function.

The header of the `display` function shows that the function expects to receive a structure pointer:

```
void display( struct employee *e_ptr  )
```

The expression in parentheses specifies what type of value the function expects. This expression is a bit complex, so let's look at each part individually. The expression `*e_ptr` indicates the function expects to receive a pointer, which it names `e_ptr`. It is preceded by

```
struct employee
```

which states what type of pointer `e_ptr` is. The **struct** keyword indicates `e_ptr` is a pointer to a structure, and the tag `employee` specifies the structure type.

The next item of interest in EMPLOY1.C is the function call that passes the structure pointer:

```
display( &jones );
```

This statement uses the address-of operator to pass the address of the `jones` structure to the `display` function. The address-of operator is not optional. Since we want the function to access the original structure—not a local copy—we must pass the structure's address.

When the `display` function executes, it creates a pointer variable named `e_ptr` and assigns to it the address passed in the function call. Now the `display` function can refer to any member of the structure indirectly through the pointer `e_ptr`. Within the `display` function, the statement

```
printf( "%s\n", e_ptr->name );
```

has the same effect that the statement

```
printf( "%s\n", jones.name );
```

has in the **main** function. Figure 9.2 illustrates the relationship between the structure pointer and structure members in EMPLOY1.C.

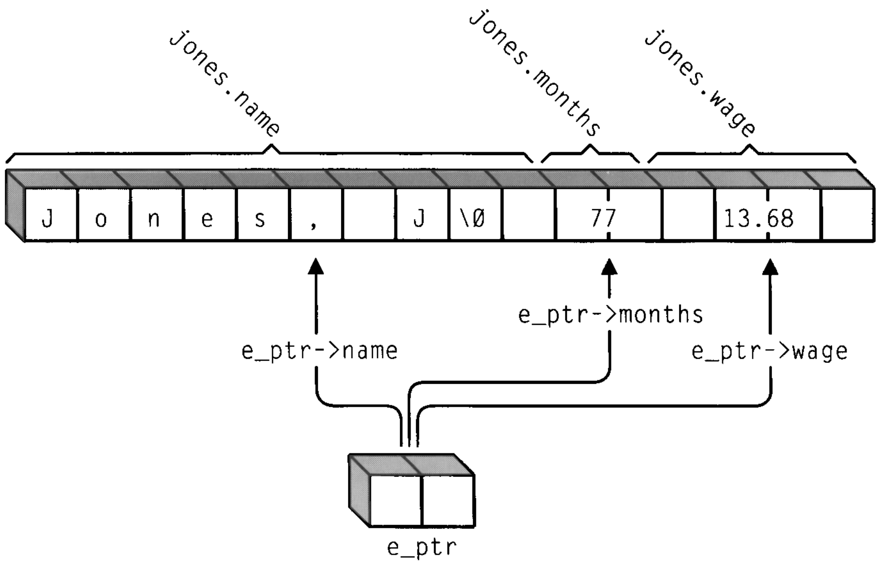


Figure 9.2 Structure Pointers in EMPLOY1.C

Just to confirm that the `display` function can access the original structure in EMPLOY1.C, try adding this statement to the end of the `display` function:

```
strcpy( e_ptr->name, "King, M" );
```

and this statement to the end of the **main** function:

```
printf( "%s\n", jones.name );
```

These changes cause EMPLOY1.C to print:

```
King, M
```

Acting indirectly through a structure pointer, the `display` function was able to change a structure defined elsewhere in the program.

Pointers to Functions

At the beginning of the previous chapter we stated that a pointer can point to any object present in memory at run time. Since functions themselves are located in memory, you can assign the address of a function to a pointer, creating a "function pointer."

A function pointer makes it possible to pass a function as a function argument.

Function pointers provide a way—in fact, the only practical way—to pass a function as an argument to another function. This permits the second function to call the first function indirectly through the pointer.

While function pointers may sound rather obscure, they have some common practical uses:

- Some QuickC run-time library functions, such as **qsort**, expect to receive a pointer to a user-defined function in your program. (Online help includes an example program that uses **qsort**.)
- Function pointers are used extensively in Windows and OS/2 Presentation Manager programs.
- Using an array of function pointers, you can create a "dispatch table." A dispatch table is a list of related functions that can be called based on some choice made at run time. It is similar to an **ON GOSUB** statement in BASIC or a call table in assembly language.

The syntax for function pointers is a bit complex, so let's start with a simple example. The FUNCPTR.C program creates a pointer to our old friend, **printf**, and calls **printf** through the pointer:

```
/* FUNCPTR.C: Demonstrate function pointers. */

#include <stdio.h>

main()
{
   int (*func_ptr) ();
   func_ptr = printf;
   (*func_ptr) ( "Curiouser and curiouser...\n" );
}
```

Here is the output from FUNCPTR.C:

```
Curiouser and curiouser...
```

This line from FUNCPTR.C declares `func_ptr` as a pointer to a function:

```
int (*func_ptr) ();
```

The declaration of a function pointer must use the same type specifier as the function it references. If the function returns a **float** value, the pointer uses type **float**, and so on. Since the **printf** function returns an **int** value showing how many characters it displays, the declaration of `func_ptr` uses the type **int**.

A function-pointer declaration must have two pairs of parentheses.

Function-pointer declarations may look complex, but all the parentheses are essential. The empty parentheses at the end of the declaration are needed to show the pointer points to a function.

The parentheses enclosing the function name itself are mandatory, too. Notice what happens if you omit them:

```
void *func_ptr(); /* Error! Not a function pointer. */
```

Instead of declaring a pointer to a function, this statement declares a function that returns a pointer—not at all what we want in FUNCPTR.C.

The next program line initializes the function pointer, assigning it the address of the **printf** function:

```
func_ptr = printf;
```

This line has two important features. First, notice the name **printf** isn't followed by parentheses, as it would be when you call **printf** directly. We want to obtain the address of **printf**, not call it.

Second, note that it's not necessary to place the address-of operator before the name **printf**. Because `func_ptr` was declared as a function pointer, the compiler knows it should use the address of **printf** here. If you like, however, you can add the address-of operator to make the statement a little more readable:

```
func_ptr = &printf;
```

The next line calls the **printf** function indirectly through the pointer `func_ptr`:

```
(*func_ptr) ( "Curiouser and curiouser...\n" );
```

Note the similarity between this statement and a normal call to **printf**. It's equivalent to this line:

```
printf( "Curiouser and curiouser...\n" );
```

To call **printf** indirectly through `func_ptr`, you supply the same arguments as when you call **printf** directly.

Passing Function Pointers as Arguments

Function pointers are usually passed as function arguments.

Like other pointers, function pointers can be passed as arguments to functions. Normally, in fact, this is the only reason to use a function pointer.

The FUNCPTR.C program in the previous section is easy to follow but not very practical. In a real program, you wouldn't go to the trouble of creating a function pointer just to call **printf** from the **main** function.

The FUNCPTR1.C program demonstrates how to pass a function pointer as an argument. It has a function named `gimme_func` that expects to be passed a function pointer:

```
/* FUNCPTR1.C: Passing function pointers as arguments. */
```

```
#include <stdio.h>

void gimme_func( void (*func_ptr) () );

main()
{
   gimme_func( puts );
   gimme_func( printf );
}

void gimme_func( void (*func_ptr) () )
{
   (*func_ptr) ( "Ausgezeichnet!" );
}
```

Here is the output from FUNCPTR1.C:

```
Ausgezeichnet!
Ausgezeichnet!
```

In the interests of brevity, the function `gimme_func` does a very simple job. It expects to receive a pointer to a function that can display a string and uses that pointer to print the string. The first call to `gimme_func` passes a pointer to the library function **puts**, and the second passes a pointer to **printf**.

Since the declaration of `gimme_func` states it takes a pointer to a function, the address-of operator is optional in a call to `gimme_func`. The following statements are equivalent:

```
gimme_func( puts );
gimme_func( &puts );
```

A Parting Word on Pointers

If you have read the previous two chapters from beginning to end, you may be suffering from a mild—or perhaps not so mild—case of information overload. Pointers have so many different uses that it's difficult to learn everything about them at once.

Don't be discouraged if some uses of pointers still aren't clear to you. The latter parts of this chapter cover some rather esoteric techniques, which you probably won't use often. When needed, however, these techniques offer some very powerful capabilities.

Like other programming concepts, pointers are best learned through practice, so use them at every sensible opportunity. Remember, you don't need to know everything about pointers in order to do *something* with them. The more you use pointers in everyday programming, the sooner all the pieces of the puzzle will fall into place.

CHAPTER

Programming Pitfalls 10

In C, as in every language, it's rare for any program to work perfectly the first time. An important part of knowing a language is recognizing what *not* to do and why certain problems occur.

This chapter describes common C programming pitfalls and how to avoid them. It is organized under broad topics, such as "Pointer Problems," with a category for miscellaneous problems at the end. The description of each error gives a code example, explains why the error occurs, and offers a solution.

Operator Problems

The most common operator problems involve operators unique to C. Others involve questions of precedence, which can cause problems in any language.

Confusing Assignment and Equality Operators

A common error is to confuse the assignment operator (=) with the equality operator (==). The mistake often occurs in decision-making statements:

```
int val = 555;
if( val = 20 )   /* Error! */
   printf( "val equals 20\n" );
```

The above code prints `val equals 20` even though it's clear `val` doesn't equal 20 when the **if** statement begins. Instead of testing whether `x` equals 20, the expression `val = 20` *assigns* the value 20 to `val`.

Remember, the single equal sign (=) performs an assignment in C. This particular assignment results in a nonzero value, so the **if** test is evaluated as true, causing the **printf** statement to execute.

To correct the problem, use the double equal sign (==) to test equality:

```
if( x == 20 )
   printf( "x equals 20\n" );
```

Once you're in the habit of using the equality operator, you might make the opposite mistake of using two equal signs where you should use only one:

```
main()
{
   int val;
   for( val == 0; val < 5; val++ )  /* Error! */
      printf( "val = %d\n", val );
}
```

Here the error appears in the initializing expression of the **for** statement. It's the reverse of what happened in the first example. Instead of assigning the value 0 to `val`, the expression `val == 0` evaluates whether or not `val` equals 0. The expression doesn't change the value of `val` at all. Since `val` is an uninitialized variable, the **for** loop is unpredictable.

Confusing Operator Precedence

Peculiar things can happen if you ignore operator precedence:

```
main()
{
   int ch;
   while( ch = getch() != '\r' )
      printf( "%d\n", ch );
}
```

Instead of assigning the result of the **getch** library-function call to `ch`, the above code assigns the value 0 to `ch` when you press the ENTER key and the value 1 when you press any other key. (The values 1 and 0 represent true and false.)

The error occurs because the inequality operator (!=) has higher precedence than the assignment operator (=). The expression

```
ch = getch() != '\r'
```

is the same as

```
ch = (getch() != '\r')
```

Both expressions compare the result of the **getch** call to the character constant **\r**. The result of that comparison is then assigned to `ch`.

For the program to work correctly, these operations must happen in the reverse order. The result of the function call must be assigned to the variable *before* the variable is compared to the constant. We can solve the problem by adding parentheses:

```
main()
{
   int ch;
   while( (ch = getch()) != '\r')
      printf( "%d\n", ch );
}
```

Parentheses have the highest precedence of any operator, so the expression

```
(ch = getch()) != '\r'
```

works correctly. It assigns the result of the **getch** call to `ch` before comparing `ch` to the constant.

The list of precedence-related errors is almost endless. Fortunately, QuickC makes it unnecessary to memorize precedence rules. To view a complete table of operator precedences, see Appendix A, "C Language Guide," and online help in the QuickC environment.

Use parentheses to avoid operator precedence problems.

When in doubt, use extra parentheses to make the order of operations absolutely clear. Extra parentheses don't degrade performance, and they can improve readability as well as minimize precedence problems.

Confusing Structure-Member Operators

Two different operators are used to access the members of a structure. Use the structure-member operator (**.**) to access a structure member directly, and the pointer-member operator (**–>**) to access a structure member indirectly through a pointer.

For instance, you may create a pointer to a structure of the `employee` type,

```
struct employee *p_ptr;
```

and initialize the pointer to point to the `jones` structure:

```
p_ptr = &jones;
```

If you use the structure-member operator to access a structure member through the pointer,

```
p_ptr.months = 78; /* Error! */
```

QuickC issues this error message:

```
C2040:     requires struct/union name
```

Use the pointer-member operator to access a structure member through a pointer:

```
p_ptr->months = 78;
```

Array Problems

The most common errors associated with arrays involve indexing errors. The problems described in this section all concern indexing errors of one form or another.

Array Indexing Errors

The first C array subscript is 0.

If you're used to a language that has different subscripting rules, it's easy to forget that the first subscript of a C array is 0 and the last subscript is 1 less than the number used to declare the array. Here's an example:

```
int i_array[4] = { 3, 55, 600, 12 };
main()
{
   int count;
   for( count = 1; count < 5; count++ )  /* Error! */
      printf( "i_array[%d] = %d\n", i_array[count] );
}
```

The **for** loop in the above program starts at `i_array[1]` and ends at `i_array[4]`. It should begin with the first element, `i_array[0]` and end at the last, `i_array[3]`. The following corrects the error.

```
for( count = 0; count < 4; count++ )
   printf( "i_array[%d] = %d\n", i_array[count] );
```

Omitting an Array Subscript in Multidimensional Arrays

Enclose each subscript in its own set of brackets.

Programmers who know QuickBASIC, QuickPascal, or FORTRAN may be tempted to place more than one array subscript in the same pair of brackets. In C, each subscript of a multidimensional array is enclosed in its own pair of brackets:

```
int i_array[2][2] = { { 12, 2 }, { 6, 55 } };
main()
{
   printf( "%d\n", i_array[ 0, 1 ] ); /* Error! */
}
```

In the preceding example, the expression

```
i_array[ 0, 1 ]
```

does not access element 0,1 of `i_array`. Here is the correct way to refer to that array element:

```
i_array[0][1]
```

Interestingly, the deviant array reference doesn't cause a syntax error. As mentioned in Chapter 6, "Operators," it's legal to separate multiple expressions with a comma operator, and the final value of such a series is the value of the right-most expression in the group. Thus, the expression

```
i_array[ 0, 1 ]
```

is equivalent to this one:

```
i_array[ 1 ];
```

Both expressions give an address, not the value of an array element.

Overrunning Array Boundaries

Since C doesn't check array subscripts for validity, you must keep track of array boundaries on your own. For instance, if you initialize a five-character array,

```
char sample[] = "ABCD";
```

and refer to a nonexistent array element,

```
sample[9] = 'X';
```

QuickC doesn't signal an error, although the second statement overwrites memory outside the array. It stores a character in element 9 of an array that contains only 5 elements.

The same problem can occur when accessing an array through a pointer:

```
char sample[] = "ABCD";
char *ptr = sample;
*--ptr = 'X';  /* Error! */
```

The code overwrites the byte in memory below the array. To avoid such problems, confine all array operations within the range used to declare the array.

String Problems

Strings are handled a little differently in C than most languages—a fact that can cause problems. The following errors are common to programs that use strings.

Confusing Character Constants and Character Strings

Remember the difference between a character constant, which has one byte, and a character string, which is a series of characters ending with a null character:

```
char ch = 'Y';
if( ch == "Y" )  /* Error! */
   printf( "The ayes have it..." );
```

The example above mistakenly compares the **char** variable `ch` to a two-character string (`"Y"`) instead of a single character constant (`'Y'`). Since the comparison is false, the **printf** statement never executes—no matter what `ch` equals.

The **if** statement needs to use single quotes. This code correctly tests whether `ch` equals the character `'Y'`:

```
char ch = 'Y';
if( ch == 'Y' )
   printf( "The ayes have it..." );
```

Forgetting the Null Character That Terminates Strings

Remember that strings end with a null character in C. If you declare this five-character array,

```
char sample[5];
```

the compiler allocates five bytes of memory for the array. If you try to store the string `"Hello"` in the array like this,

```
strcpy( sample, "Hello" );
```

you'll overrun the array's bounds. The string `"Hello"` contains six characters (five letters and a null character), so it's one byte too big to fit in the `sample` array. The **strcpy** overwrites one byte of memory outside the array's storage.

It's easy to make this error when allocating memory for a string, too:

```
char str[] = "Hello";
char *ptr;
ptr = malloc( strlen( str ) );  /* Error! */
if( ptr == NULL )
   exit( 1 );
else
   strcpy( ptr, str );
```

This time the error occurs in the call to the **malloc** function, which allocates memory to a pointer prior to a string copy. The **strlen** function returns the length of a string not including the null character that ends the string. Since the amount of memory allocated is one byte too small, the **strcpy** operation overwrites memory, just as in the previous example.

To avoid the problem, add 1 to the value returned by **strlen**:

```
ptr = malloc( strlen( str ) + 1 );
```

Forgetting to Allocate Memory for a String

If you declare a string as a pointer, don't forget to allocate memory for it. This example tries to create a **char** pointer named `ptr` and initialize it with a string:

```
main()
{
   char *ptr;
   strcpy( ptr, "Ashby" );  /* Error! */
}
```

The pointer declaration `char *ptr;` creates a pointer variable but nothing else. It allocates enough memory for the pointer to store an address but doesn't allocate any memory to store the object to which `ptr` will point. The **strcpy** operation in the next line overwrites memory by copying the string into an area not used by the program.

One way to allocate memory is by declaring a **char** array large enough to hold the string:

```
main()
{
   char c_array[10];
   strcpy( c_array, "Randleman" );
}
```

You can also call the **malloc** library function to allocate memory at run time:

```
#define BUFFER_SIZE 30
#include <malloc.h>

main()
{
   char *ptr;
   if( ptr = (char *) malloc( BUFFER_SIZE ) )
   {
      strcpy( ptr, "Duvall" );
      printf( ptr );
      free( ptr );
   }
}
```

Pointer Problems

Every experienced C programmer has a collection of favorite pointer-induced bugs. Pointer errors can wreak havoc because pointers can change the contents of any addressable memory location. If a pointer writes to an unexpected address, the results can be disastrous.

Using the Wrong Address Operator to Initialize a Pointer

If you're still learning about pointers, it's easy to forget which address operator to use when initializing a pointer variable. For example, you might want to create a pointer to a simple **int** variable:

```
int val = 25;
int *ptr;
ptr = val; /* Error! */
```

The code above doesn't initialize `ptr` correctly. Instead of assigning to `ptr` the address of `val`, the statement

```
ptr = val;
```

tries to assign `ptr` the contents of `val`, causing an error message:

```
warning C4047: '=' : different levels of indirection
```

Because `val` is an **int** variable, its contents can't form a meaningful address for `ptr`. You must use the address-of operator to initialize `ptr`:

```
ptr = &val;
```

Here's another pointer initialization error:

```
int val = 25;
int *ptr;
*ptr = &val; /* Error! */
```

The last line doesn't initialize `ptr` to point to the variable `val`. The expression to the left of the equal sign, `*ptr`, stands for the object `ptr` points to. Instead of assigning `ptr` the address of `val`, the line tries to assign the address of `val` to the place where `ptr` points. Because `ptr` has never been initialized, the assignment triggers a run-time error:

```
run-time error R6001
-null pointer assignment
```

Here is the correct way to initialize this pointer:

```
ptr = &val;
```

Declaring a Pointer with the Wrong Type

You should make sure the type used to declare a pointer matches the type of data object it points to:

```
main()
{
   int *ptr;
   .
   .
   .
   float val = 3.333333;
   ptr = &val; /* Error! */
   printf( "val = %f\n", *ptr );
}
```

The program declares `ptr` as a pointer to an **int**. Later on, forgetting what type we used when declaring `ptr`, we assign it the address of the floating-point variable `val`.

Declaring a pointer with the wrong type can cause unwanted type conversions.

Since C allows you to assign any address to a pointer, the assignment doesn't cause an error. But accessing `val` through `ptr` creates problems. Because `ptr` is declared as a pointer to an **int**, the compiler does a type conversion on the **float** it points to, converting the **float** value to an **int**. The output is garbage:

```
val = 11242989923343410000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000.000000
```

The following program cures the error by declaring `ptr` as a pointer to a **float** data type:

```
main()
{
   float *ptr;
   float val = 3.333333;
   ptr = &val;
   printf( "%f\n", *ptr );
}
```

Now it gives the correct output:

```
val = 3.333333
```

Using Dangling Pointers

A "dangling pointer" is one that points to a memory area no longer in use by your program. Dangling pointers, like uninitialized pointers, can be very dangerous to use.

For instance, say you allocate a block of memory with the **malloc** library function:

```
#define BUFSIZE 1000
char *ptr;
if( ptr = (char *) malloc( BUFSIZE ) )
   /* do something */ ;
```

After the memory block has been allocated with **malloc**, the pointer `ptr` points to a valid data object. Once you're done using allocated memory, you normally return it to the heap:

```
free( ptr );
```

After you free the memory it points to, `ptr` is a dangling pointer. It still points to a valid machine address, but that address is no longer in use by the program. You shouldn't use the pointer at this stage, just as you shouldn't use it before it has been initialized.

Dangling pointers can also be created by a function that returns a pointer to a local variable:

```
int *boo_boo( void )
{
   int object;
   .
   .
   .
   return &object; /* Error! */
}
```

The `boo_boo` function returns the address of the local variable `object`, forgetting the storage for `object` is no longer part of the program after the function ends.

Here's a variant of the previous example involving a string pointer:

```
char *boo_boo( void )
{
   char *c_ptr;
   c_ptr = "Hello";
   .
   .
   .
   return c_ptr; /* Error! */
}
```

Since the string constant `"Hello"` is local to the function, it evaporates when the function ends, leaving the pointer `c_ptr` dangling.

Library-Function Problems

Once you've learned enough about C to write practical programs, you can begin to explore the rich function library supplied with QuickC. This section outlines a few common problems related to using library functions. Again, you can use on-line help to get information about specific library functions.

Failing to Check Return Values from Library Functions

Always check library function return values.

Almost all library functions return some value—either the result of processing or an error code showing success or failure. You should always check library-function return values, even if you're confident of the result.

This rule is critical when calling a library function such as **malloc**, which allocates memory at run time:

```
char *ptr;
ptr = (char *) malloc( BUFSIZE );   /* Error! */
```

If the call to **malloc** fails, the pointer `ptr` is assigned a null (0) value. Using `ptr` under these circumstances can overwrite unexpected memory addresses or cause a run-time error. The following code checks the return value from **malloc**:

```
#define NULL 0
#define BUFSIZE 32768
    .
    .
    .
char *ptr;
if( (ptr = (char *) malloc( BUFSIZE ) ) != NULL )
{
   printf( "Copacetic.\n" );
   /* Do something useful... */
}
else
   printf( "Not enough memory!\n" );
```

Duplicating Library-Function Names

There are so many functions in the QuickC run-time library that it's sometimes difficult to avoid duplicating function names. For instance, if you write a function that reads data from a buffer, the name `read` may strike you as short and descriptive.

The only problem is that **read** is the name of a QuickC library function. A program that defines its own `read` function may work correctly at first, but if you later include the header file that declares the **read** library function,

```
#include <io.h>
```

then redefinition errors occur. You can't use the same name for two different functions. The solution here is to rename the user-defined function.

Use online help to check for function-name conflicts.

QuickC's online help lets you check for such name conflicts on the spot. Put the cursor on the function name you wish to use, then press F1. If the name is already used for a library function, online help displays information about the function. If the name isn't in online help, it's not used in the QuickC function library and is a safe choice.

Unless you're writing your own library functions, it's a good rule to avoid declaring names that begin with an underscore (_), since many of the system-defined names in QuickC start with that character. (Non-ANSI library functions begin with a single underscore. Predefined identifiers such as **__TIME__** start with two underscores, and routines internal to the C run-time library can begin with either one or two underscores.)

Forgetting to Include Header Files for Library Functions

Because they contain needed function prototypes, it's important to include the correct header files when using QuickC library functions:

```
main()
{
   double val = sqrt( (double) 10 );
   printf( "square root of 10 = %le\n", val );
}
```

The program above calls the library function **sqrt**, which calculates a square root. Most of the program is correct. When passing the value 10 to **sqrt**, it casts the argument as a **double**, the type **sqrt** expects. The return value from **sqrt** is assigned to a **double** variable, too.

Unfortunately, the program still gives the wrong output. The square root of 10 is not 171 (1.710000e+002 in exponential notation):

```
square root of 10 = 1.710000e+002
```

Function prototypes can prevent unexpected type conversions.

Because the program has no prototype for the **sqrt** function, **sqrt** has the **int** return type by default. The value returned by **sqrt** undergoes an unexpected type conversion—from type **double** to **int**—and becomes garbage.

This problem is easily solved. Simply include the standard header file that contains the prototype for **sqrt**:

```
#include <stdio.h>
#include <math.h>
main()
{
   double val = sqrt( (double) 10 );
   printf( "square root of 10 = %le\n", val );
}
```

Now the program works correctly:

```
square root of 10 = 3.162278e+000
```

If you're not sure which header file a library function needs, take advantage of QuickC's online help. (Put the cursor on the function name and press F1.) If the function needs a header file, the file name appears in an **#include** directive above the function prototype.

Omitting the Address-Of Operator When Calling scanf

Don't forget to put the address-of operator in front of arguments when using the **scanf** library function (the **scanf** function accesses keyboard input; see Chapter 11, "Input and Output"):

```
main()
{
   int val;
   printf( "Type a number: " );
   scanf( "%d", val ); /* Error! */
   printf( "%d", val );
}
```

When the program calls **scanf**, it omits the address-of operator that should precede the second argument:

```
scanf( "%d", val );  /* Error! */
```

The **scanf** function expects to be passed a pointer to a variable (in this case, a pointer to `val`) so it can assign an input value to the variable. But because the address-of operator is missing, the program passes the value of `val`, not its address.

Instead of storing an input value in `val` as intended, **scanf** uses the uninitialized value of `val` as a pointer and assigns the input value to an unpredictable address. As a result, `val` remains uninitialized and the program overwrites memory elsewhere—two very undesirable events.

Here is the correct way to call **scanf** in this program:

```
scanf( "%d", &val );
```

Macro Problems

Function-like macros—macro definitions that take arguments—share many of the advantages of functions. They can cause unwanted side effects, however, if you fail to put parentheses around their arguments or carelessly supply an argument that uses an increment or decrement operator.

Omitting Parentheses from Macro Arguments

A macro definition that doesn't enclose its arguments in parentheses can create precedence problems:

```
#include <stdio.h>

#define FOURX(arg)  ( arg * 4 )

main()
{
   int val;
   val = FOURX( 2 + 3 );
   printf( "val = %d\n", val );
}
```

The `FOURX` macro in the program multiplies its argument by 4. The macro works fine if you pass it a single value, as in

```
val = FOURX( 2 );
```

but returns the wrong result if you pass it this expression:

```
val = FOURX( 2 + 3 );
```

QuickC expands the above line to this line:

```
val = 2 + 3 * 4;
```

Use parentheses to avoid precedence problems in macros.

Because the multiplication operator has higher precedence than the addition operator, this line assigns `val` the value 14 (or 2 + 12) rather than the correct value 20 (or 5 * 4).

You can avoid the problem by enclosing the macro argument in parentheses each time it appears in the macro definition:

```
#include <stdio.h>

#define FOURX(arg)  ( (arg) * 4 )

main()
{
   int val;
   val = FOURX(2 + 3);
   printf( "val = %d\n", val );
}
```

Now the program expands this line

```
val = FOURX(2 + 3);
```

into this one:

```
val = (2 + 3) * 4;
```

The extra parentheses assure that the addition is performed before the multiplication, giving the desired result.

Using Increment and Decrement Operators in Macro Arguments

If a function-like macro evaluates an argument more than once, you should avoid passing it an expression that contains an increment or decrement operator:

```
#include <stdio.h>
#define ABS(value)  ( (value) >= 0 ? (value) : -(value) )

main()
{
   int array[4] = {3, -20, -555, 6};
   int *ptr = array;
   int val, count;
   for( count = 0; count < 4; count++ )
   {
      val = ABS(*ptr++); /* Error! */
      printf( "abs of array[%d] = %d\n", count, val );
   }
}
```

The program uses the ABS macro that was used to explain macros in Chapter 7, "Preprocessor Directives." The macro returns the absolute value of the argument you pass to it.

The goal in this program is to display the absolute value of every element in array. It uses a **for** loop to step through the array and a pointer named ptr to access each array element in turn. Instead of the output you would expect,

```
abs of array[0] = 3
abs of array[1] = 20
abs of array[2] = 555
abs of array[3] = 6
```

the program gives this output:

```
abs of array[0] = -20
abs of array[1] = -6
abs of array[2] = 8307
abs of array[3] = 24864
```

(The last two array values may differ if you run the program. They are the contents of memory not used by the program.)

The error occurs in this line,

```
val = ABS(*ptr++); /* Error! */
```

which QuickC expands as shown here:

```
val = ( (*ptr++) >= 0 ? (*ptr++) : -(*ptr++) ); /* Error! */
```

Because it uses the conditional operator, the ABS macro always evaluates its argument at least twice. This isn't a problem when the argument is a constant or simple variable. In the example, however, the argument is the expression *ptr++. Each time the macro evaluates this expression, the increment operator takes effect, causing ptr to point to the next element of array.

The first time the program invokes the macro, ptr points to the first array element, array[0]. Since this element contains a nonnegative value (3) the macro evaluates the argument twice. The first evaluation takes the value that ptr points to and then increments ptr. Now ptr points to the second element, array[1]. The second evaluation takes the value of array[1] and increments ptr again.

The first macro invocation not only returns an incorrect value (–20, the value of array[1]). It also leaves ptr pointing to the third array element, making the results of later invocations unpredictable. (The pointer eventually moves past the last element of array and points to unknown data.)

To avoid the problem, don't use the increment or decrement operators in arguments you pass to a macro. This revision removes the error by incrementing `ptr` in the **for** statement instead of the macro invocation:

```
#include <stdio.h>
#define ABS(value)  ( (value) >= 0 ? (value) : -(value) )

main()
{
   int array[4] = {3, -20, -555, 6};
   int *ptr = array;
   int val, count;
   for( count = 0; count < 4; count++, ptr++ )
   {
      val = ABS(*ptr);
      printf( "abs of array[%d] = %d\n", count, val );
   }
}
```

This advice applies generally to QuickC library routines as well as macros you write. Remember, some run-time library routines are implemented as macros rather than C functions. If you're not sure whether a library routine is actually a macro, look it up in online help.

Miscellaneous Problems

This section describes C programming problems that don't fit into any convenient category.

Mismatching if and else Statements

In nested **if** statements, each **else** is associated with the closest preceding **if** statement that does not have an **else**. Although indentation can make nested constructs more readable, it has no syntactical effect:

```
if( val > 5 )
   if( count == 10 )
      val = sample;
else
   val = 0;
```

The indentation suggests that the **else** associates with the first **if**. In fact, the **else** is part of the second **if**, as shown more clearly here:

```
if( val > 5 )
   if( count == 10 )
      val = sample;
   else
      val = 0;
```

The **else** is part of the second **if** statement—the closest preceding **if** that doesn't have a matching **else**. To tie the **else** to the first **if**, you must use braces:

```
if( val > 5 )
{
   if( count == 10 )
      val = sample;
}
else
   val = 0;
```

Indentation makes programs easier to read, but is ignored by the compiler.

Now the **else** belongs with the outermost **if**. Remember, indentation is meaningful only to humans. The compiler relies strictly on punctuation when it translates the source file.

Misplacing Semicolons

Misplaced semicolons can cause subtle bugs:

```
#include <stdio.h>

main()
{
   int count;
   for( count = 0; count < 500; count++ ); /* Error! */
   {
      printf( "count = %d\n", count );
      printf( "And the beat goes on...\n" );
   }
}
```

You might expect the program to print the value of `count` 500 times, but this is all it prints:

```
count = 500
And the beat goes on...
```

The culprit is the extra semicolon immediately after the parentheses of the **for** statement. Its effect is more evident if we reformat the statement:

```
#include <stdio.h>

main()
{
   int count;
   for( count = 0; count < 500; count++ )
      ; /* Null statement */
   {
      printf( "count = %d\n", count );
      printf( "And the beat goes on...\n" );
   }
}
```

Instead of printing the value of `count` 500 times, the program executes the null statement (;) 500 times. Null statements are perfectly legal in C, so the compiler has no way to tell this is a mistake.

Since the null statement is interpreted as the loop body, the **printf** statements inside curly braces are interpreted as a statement block and executed once. Statement blocks usually appear as part of a loop, function definition, or decision-making statement, but it's legal to enclose any series of statements in braces.

The program works as intended if you remove the extra semicolon:

```
#include <stdio.h>

main()
{
   int count;
   for( count = 0; count < 500; count++ )
   {
      printf( "count = %d\n", count );
      printf( "And the beat goes on...\n" );
   }
}
```

Here's another one. If you know QuickPascal, you might be tempted to put a semicolon after the parentheses of a function definition:

```
void func( void );

void func( void ); /* Error! No semicolon here. */
{
   printf( "C is not Pascal\n" );
}
```

The function header causes a syntax error. While a function declaration requires a semicolon after its parentheses, a function definition does not. This code corrects the error:

```
void func( void );

void func( void )
{
   printf( "C is not Pascal\n" );
}
```

Omitting Double Backslashes in DOS Path Specifications

Because C uses the backslash (\) as an escape character, it's easy to create garbled path specifications:

```
fp = fopen( "c:\temp\bodkin.txt", "w" );
```

At first glance, the path specification in the string

```
"c:\temp\bodkin.txt"
```

looks good because that's how you would type it on the DOS command line. In a quoted string, however, the backslash is interpreted as an escape character. In this string the sequences **\t** and **\b** are interpreted as the tab and backspace character, respectively, garbling the path and file name. Even if the indicated file exists, this call to **fopen** is sure to fail.

In a quoted string the escape sequence for a backslash character is a double backslash (\\). This statement solves the problem:

```
fp = fopen( "c:\\temp\\bodkin.txt", "w" );
```

Omitting break Statements from a switch Statement

Don't forget to include **break** statements when using the **switch** statement:

```
switch( ch )
{
   case 'e':
      printf( "Bye bye\n" );
      break;
   case 'l':
      printf( "Loading the file\n" );
      load_file( fp );
      break;
```

```
   case 's':
      printf( "Saving the file\n" );
      write_file( fp );  /* Error! Missing break. */
   case 'd':
      printf( "Deleting the file\n" );
      kill_file( fp );
      break;
   default:
      break;
}
```

In this code a **break** statement is missing from the statements following the third case label (the statements that print `Saving the file`). After those statements execute, execution falls through to the next case label, deleting the newly saved file.

To avoid this problem, place a **break** at the end of every case item:

```
case 's':
   printf( "Saving the file.\n" );
   write_file( fp );
   break;
```

It's legal, of course, to write a program in which execution deliberately falls through from one case label to the next. In such cases you may want to add a comment to prevent confusion.

Mixing Signed and Unsigned Values

If you explicitly compare two values of different types, the compiler normally catches the error. Some type mismatches aren't easy to spot, however, even for humans:

```
#define CHARVAL '\xff'

main()
{
   unsigned char uc;
   uc = CHARVAL;
   if( uc == CHARVAL )
      printf( "Eureka!" );
   else
      printf( "Oops..." );
}
```

The program prints `Oops...` which probably wasn't expected. The comparison between `CHARVAL` and `uc` is false even though both are clearly **char** values.

The answer lies in the way the compiler converts **signed** and **unsigned char** values into **int** values for internal use. The **#define** directive,

```
#define CHARVAL '\xff'
```

defines `CHARVAL` as the constant 0xff. Since no sign is specified, the compiler treats the constant as a **signed char** value by default. When it converts the **char** to an **int** for internal use, as it does all character values, the compiler extends the value's sign. The result is an **int** with the value 0xffff.

The variable `uc` undergoes the same internal conversion, with an important difference. Since `uc` is explicitly declared as **unsigned**, its value is converted to an **int** value of 0x00ff.

When the two **int** values are compared, the result is false (0xffff does not equal 0x00ff). One solution is to explicitly cast `CHARVAL` to the desired type:

```
#define CHARVAL (unsigned char)'\xff'
```

Now the compiler compares two **unsigned char** values, giving the desired result. Another solution is to make `CHARVAL` an **int** instead of a **char** constant:

```
#define CHARVAL 0xff
```

Both solutions give the desired result, although the second is slightly less efficient. It creates word-size, rather than byte-size, machine-code instructions.

PART 2

Using C

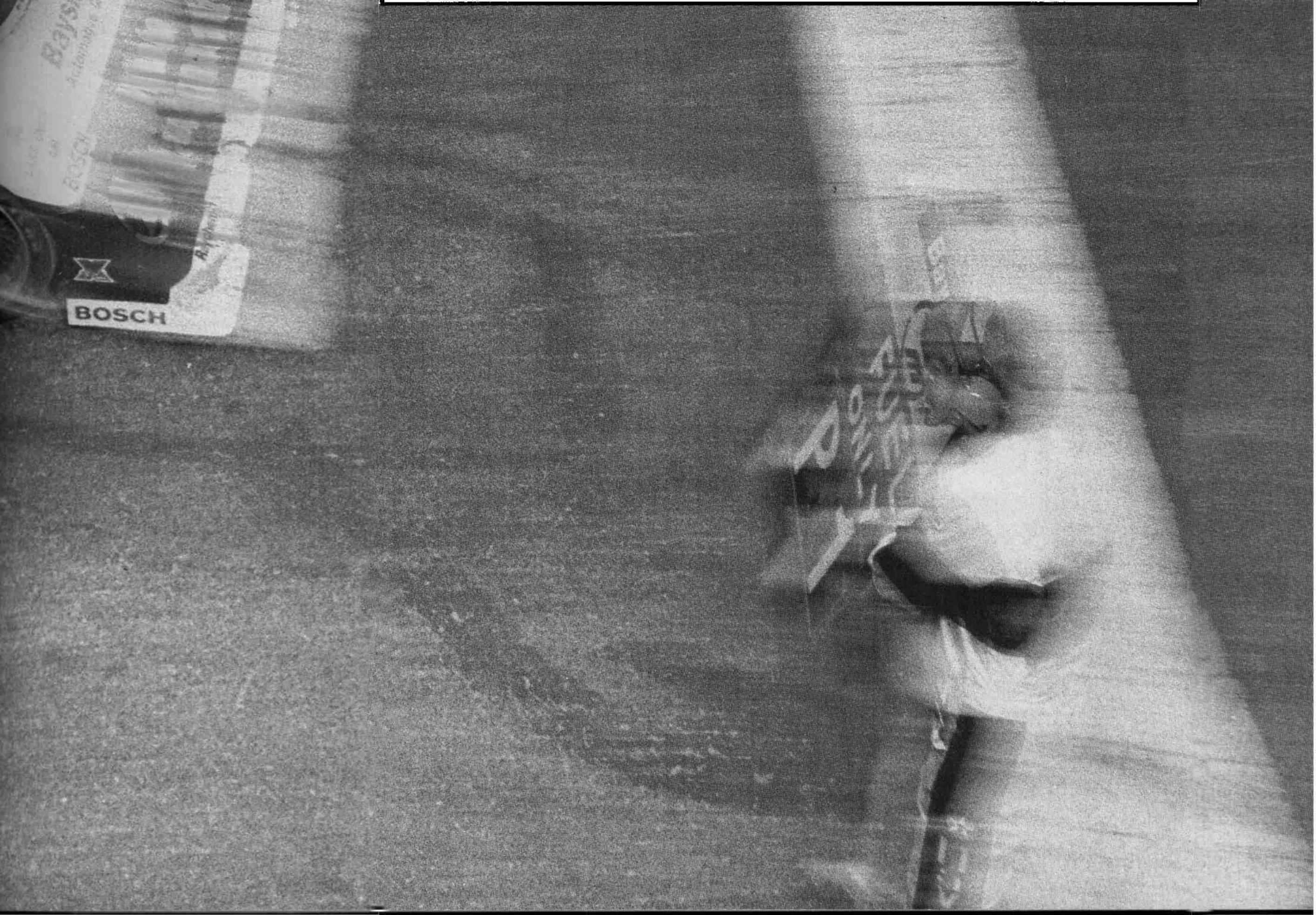
CHAPTERS


Bayside
86

Using C

Part 2 of *C for Yourself* is called "Using C" and should be read after you are familiar with basic C concepts. It covers practical topics that make it possible for you to write real programs. The features discussed in these chapters are provided in the QuickC run-time library, which, as you may recall from Part 1, is not part of the C language itself.

While Part 1 was designed to be read sequentially, Part 2 is topical. So you don't need to read its chapters in any particular order. If you are new to C, however, it is recommended that you begin with Chapter 11, "Input and Output," which describes how to read and write data, and process files. Similarly, if you're not familiar with QuickC graphics, you should read Chapters 13–15 in order.



CHAPTER 11

Input and Output

The first part of this book explored the fundamentals of the C language. In the second part (starting with this chapter), the topics include more complex and powerful functions: accessing disk files, creating high-resolution graphics, creating graphs, manipulating fonts, and adding assembly-language routines to your C programs.

Program examples in previous chapters used **printf** to print to the screen. In this chapter, we'll cover **printf** in more detail, moving on to other I/O functions such as **fprintf**, which prints to a file, instead of to the screen.

This chapter covers three broad topics: keyboard and screen input/output (I/O), reading and writing standard disk files, and low-level disk access. It also introduces several common string-handling functions.

Input and Output Streams

Books about C often refer to "input streams" and "output streams." A stream is a sequence of bytes flowing into the program (input) or flowing out (output). The data might have originally come from the keyboard, a modem, a disk file, or some other peripheral device. The outgoing data might be sent out to the screen, a modem, or a disk file.

Thus, when you see a phrase such as "opening a stream," it means opening a line of communication to the disk drive or to some other peripheral.

Peripherals and files are called "streams" in C.

The five streams always open and available for input or output are shown in Table 11.1.

Table 11.1 Standard I/O Streams

Name	Stream
stdin	Standard input (keyboard)
stdout	Standard output (screen)
stderr	Standard error channel (screen)
stdprn	Standard printer (parallel port)
stdaux	Standard auxiliary device (serial port)

Screen and Keyboard I/O

Imagine an application program that doesn't ever send output to the screen or accept input from the keyboard. It's possible to write such a program, but it's unlikely you'd ever want to.

In most situations, you need to display various kinds of data on the screen and to accept input from the keyboard. "Manipulating and Printing Strings" introduces the functions commonly used to communicate back and forth.

Manipulating and Printing Strings

Always pass at least one string to the printf function.

Previous chapters have used the **printf** function to display results on the screen. By now you should be accustomed to how it works. There's one rule you must always follow when using **printf**: pass it at least one format string, which may be a literal string or a pointer to a string. The string may or may not include format specifiers, which are defined below.

The **printf** function always prints to the **stdout** device. Unless the output has been redirected, the standard output device is the screen.

The following program illustrates some typical ways to manipulate strings and to print them:

```
/* PRTSTR.C: Print strings. */

#include <stdio.h>
#include <string.h>

main()
{
   char aline[80], more[80];
   char *strptr;

   /* aline = "Another line."; */
   /* Note: This causes a compiler error */

   strcpy( aline, "Another line." );
   strcpy( more, aline );
   strptr = aline;
   strcat( aline, "dog" );
   printf( "A line of text." );
   printf( aline );
   printf( more );
   printf( strptr );
}
```

The declarations come first:

```
char aline[80], more[80];
char *strptr;
```

The variables `aline` and `more` are arrays of characters. In this program, they act as strings. Although these arrays have 80 characters each (numbered 0–79), the maximum string length is 79 characters, because strings must end with a null character. The variable `strptr` is a pointer to a string.

If you've previously programmed in BASIC, you might expect to use the equal sign to assign a value to a string variable. The program won't compile if you remove the comment symbols from the following line:

```
/* aline = "Another line."; */
```

Faced with this line, QuickC prints the error message:

```
2106: '=' : Left operand must be lvalue
```

(an "lvalue" is a value allowed on the left side of an equal sign).

Use the strcpy function—not the equal sign—to copy a string.

You can use the equal sign to assign a value to a numeric variable. When you're using strings, however, you almost always use the library function **strcpy**, which copies a string to a character array from either a string constant or another array:

```
strcpy( aline, "Another line." );
strcpy( more, aline );
```

The **strcpy** function makes an exact copy of a string. The first argument is the address of the destination string. The second is the address of the source string. The first **strcpy** above copies `"Another line."` to the `aline` string. The second copies `aline` to `more`.

Note that the first argument must be the address of an array, but the second is either a string constant (enclosed in quotation marks) or the address of a character array.

The 80-character arrays have more than enough room for the 13 characters of `"Another line."` and a null character. In your own programs, you should be aware of the declared size of an array and avoid overrunning the bounds of the array. See Chapter 10, "Programming Pitfalls," for more information about this programming mistake.

It is possible to assign the address of a string to a pointer:

```
strptr = aline;
```

Notice that both `strptr` and `aline` point to the same string. There's one object in memory, but it has two different names. If `aline` changes, the same change occurs in the string referenced by `strptr`, because they're the same string. Below, the word `"dog"` is added to the end of the string `aline`:

```
strcat( aline, "dog" );
```

The **strcat** function concatenates one string to the end of a second string. In the line above, both `aline` and the string referenced by `strptr` have been changed from `"Another line."` to `"Another line.dog"`.

Now four **printf** statements execute:

```
printf( "A line of text." );
printf( aline );
printf( more );
printf( strptr );
```

The screen should look like this:

```
A line of text.Another line.dogAnother line.Another line.dog
```

To the first **printf** we passed a string constant. To the other three we passed names of strings. Concatenating `aline` and `"dog"` also affected the string referenced by `strptr`, because they both point to the same string in memory. The contents of `more` weren't affected, however, because the **strcpy** function makes a complete and unique copy of the source string at the memory location referenced by `more`.

Unfortunately, the strings ran together. As we saw in Chapter 1, "Anatomy of a C Program," **printf** is unlike QuickBASIC's **PRINT** command or Pascal's **Writeln** procedure in one respect: it does not automatically move the cursor to the beginning of the next line. You need to include the newline character (**\n**), which is one of a series of available escape codes discussed in Chapter 4, "Data Types." The program below includes a few examples of escape codes, each of which begins with the backslash character:

```
/* PRTESC.C: Print escape characters \",\n, and \t. */

#include <stdio.h>
#include <string.h>

main()
{
   char b[80];
   int i,j;

   strcpy( b, "and seven years ago\n" );
   printf( "\"Four score\n" );
   printf( b );
   printf( "\tone tab\n\t\ttwo tabs\n\t\t\tthree tabs\n" );
   i = sizeof( b );
   j = strlen( b );
   printf( "Size is %d\nLength is %d.\n", i, j );
}
```

If you compile and run the PRTESC.C program, the following text prints on the screen:

```
"Four score
and seven years ago
        one tab
                two tabs
                        three tabs
Size is 80
Length is 20.
```

To print a newline character in a string, type a backslash and the letter n (**\n**). For a quotation mark, use **\"**. For tabs, use **\t**. Escape sequences can appear anywhere within a string:

```
printf( "\tone tab\n\t\ttwo tabs\n\t\t\tthree tabs\n" );
```

You'll find complete lists of escape characters in Appendix A, "C Language Guide," and in online help.

Finding the Size

The last call to **printf** in PRTESC.C provides two pieces of information: the size of the character array and the length of the string inside the array.

The variable `b` was declared to be an 80-character array, but the string inside `b` contains only 20 characters; it holds 19 letters plus one newline character. Although typing **\n** takes two characters, it's stored in memory as one character—the ASCII value 10. As we'll see later in this chapter, the newline character is sometimes expanded to two characters (a carriage return and a linefeed) when it is written to disk. But while it's in memory, it's a single character.

The sizeof operator examines array size; the strlen function returns the length of a string.

There are two methods available to find the size of arrays and strings. The **sizeof** operator returns the size (in bytes) of an identifier or type. The string-handling function **strlen** counts the number of characters in a string, up to but not including the null that marks the end of the string:

```
i = sizeof( b );
j = strlen( b );
printf( "Size is %d\nLength is %d.\n", i, j );
```

The final line of the program PRTESC.C prints out two integer values, which follow the format string. When **printf** evaluates the format string, it substitutes the two values for the **%d** specifiers:

```
Size is 80
Length is 20
```

The **sizeof** operator is part of the C language. In this example, it evaluates to the value 80, which is the size of the array. The **strlen** function is a library function for measuring strings (up to, but not including the null at the end). It returns a 20 because that's the length of the string.

Printing Numeric Values

The printf format string may hold one or more format specifiers.

We've seen how **printf** requires at least one string (or a pointer to a string). To print variables and values, place a comma and the name of the variable or value after the format string. Then, within the format string, include a format specification. See Table 11.2.

Table 11.2 Common Format Specifications

Specification	Format
%c	Print a character
%d	Print a decimal integer
%f	Print a floating-point number
%i	Print a decimal integer (same as **%d**)
%s	Print a string
%u	Print an unsigned integer
%x	Print in hexadecimal format

The percent sign (**%**) always marks the beginning of a format specification. The letters **c**, **d**, **f**, **i**, **s**, **u**, and **x** are called the "type." Between the percent sign and the type, you may include optional specifications for flags, width, or precision values.

At the very least, you must include the type, as in the program below:

```
/* NFORMAT.C: Print numbers and a string. */

#include <stdio.h>
#include <string.h>

main()
{
   int    a = -765,
          b = 1,
          c = 44000,
          d = 33;
   float  e = 1.33E8,
          f = -0.1234567,
          g = 12345.6789,
          h = 1.0;
   char   i[80];

   strcpy( i, "word 1, word 2, word 3, word 4, word 5" );

   printf( "Unformatted:\n%d %d %d %d\n", a, b, c, d );
   printf( "%f %f %f %f\n", e, f, g, h );
   printf( "%s\n", i );
}
```

The output looks like this:

```
Unformatted:
-765 1 -21536 33
133000000.000000 -0.123457 12345.678711 1.000000
word 1, word 2, word 3, word 4, word 5
```

If you carefully compare NFORMAT.C with its output, you'll notice some unexpected results. For example, the variable `c`, which was initialized to 44000, has somehow changed to –21536.

The **%d** format specification applies to signed integers in the range –32768 to +32767. The value of `c` (44000) is outside that range, but still within the realm of unsigned integers, which can hold values up to +65535. The proper format specification would be **%u** (where `u` represents the unsigned type).

Two of the floating-point values have changed, too. The **%f** specification defaults to 6 digits of precision to the right of the decimal point. The value of `f` (.1234567) is therefore rounded off to a precision of 6 digits: .123457. Also, the limitations of floating-point accuracy transform the value of `g` from 12345.6789 to 12345.678711. If you modify the program, changing the `float` declarations to `double`, the second problem disappears. The variable `g` prints correctly as 12345.67.

Between the % and the type character, you may include two numbers separated by a period. The first number is called the "width;" the second is the "precision." The width and precision affect integers, floating-point numbers, and strings in different ways. For example, we could specify a width of 2 and precision of 3 for each of the above variables:

```
printf( "\nWidth 2, Precision 3:\n" );
printf( "%2.3d %2.3d %2.3u %2.3d\n", a, b, c, d );
printf( "%2.3f %2.3f %2.3f %2.3f\n", e, f, g, h );
printf( "%2.3s\n", i );
```

(Note that the variable `c` has a format specifier of `%2.3u` instead of `%2.3d`.) The screen displays the following lines:

```
Width 2, Precision 3:
-765 001 44000 033
133000000.000 -0.123 12345.679 1.000
wor
```

For integers, the precision of 3 causes at least 3 digits to print, preceded by leading zeros. For floating-point numbers, the precision of 3 truncates fractions to 3 digits to the right of the decimal point. For strings, the precision of 3 causes only 3 characters to print. The string output is truncated to the right. Numbers are never truncated, however.

We can change the width to 8 and the precision to 1:

```
printf( "\nWidth 8, Precision 1:\n" );
printf( "%8.1d %8.1d %8.1u %8.1d\n", a, b, c, d );
printf( "%8.1e %8.1f %8.1f %8.1f\n", e, f, g, h );
printf( "%8.1s\n", i );
```

We made an additional modification by printing the variable e as an **%e** type instead of an **%f** type. This prints the value of e (1.33E8) in exponential format:

```
Width 8, Precision 1:
    -765        1    44000       33
1.3e+008     -0.1  12345.7      1.0
       W
```

The width controls the printing area: all 3 variable types are printed in fields 8 characters wide. The precision of 1 affects different data types in different ways: the integers print at least 1 digit; the floating-point numbers print only the first number to the right of the decimal point; and the string prints as the first character only. Each value prints flush right in its field.

Between the % and the width, you may also insert a flag. The plus flag (+), for example, forces numbers to print with a leading sign:

```
printf( "\nForced signs, Width 10, Precision 2:\n" );
printf( "%+10.2d %+10.2d %+10.2u %+10.2d\n", a, b, c, d );
printf( "%+10.2e %+10.2f %+10.2f %+10.2f\n", e, f, g, h );
printf( "%+10.2s\n", i );
```

Note that the plus flag has no effect on strings or on unsigned integers:

```
Forced signs, Width 10, Precision 2:
      -765        +01      44000        +33
+1.33e+008      -0.12  +12345.68      +1.00
        Wo
```

Another flag is the number 0, which forces leading zeros to print within the limits of the width. If you only specify the width, the system default is used for the precision. You can use the type **%x** to represent hexadecimal; it displays the letters a–f in lowercase. If you prefer uppercase, you can use **%X** instead.

```
printf( "\nHexadecimal, Forced Zeros, Width 6:\n" );
printf( "%06x %06x %06x %06x\n", a, b, c, d );
```

The **printf** statements above display these lines:

```
Hexadecimal, Forced Zeros, Width 6:
00fd03 000001 00abe0 000021
```

For strings, the width and precision specifiers describe the field width and the number of characters printed. Note the minus sign in the final line, which forces the truncated string to print from the left:

```
printf( "\nWidth 40, Precision 10:\n" );
printf( "%40.10s\n", i );

printf( "\nWidth 40, Precision 20:\n" );
printf( "%40.20s\n", i );

printf( "\nFlush left, Width 40, Precision 20:\n" );
printf( "%-40.20s\n", i );
```

The lines are displayed on the screen as follows:

```
Width 40, Precision 10:
                              word 1, wo

Width 40, Precision 20:
                    word 1, word 2, word

Flush left, Width 40, Precision 20:
word 1, word 2, word
```

Using scanf for Keyboard Input

Pass a variable address to scanf, not a variable value.

While **printf** is the most widely used output function, **scanf** is the most popular for input. The arguments and format strings passed to **scanf** resemble the arguments for **printf**, except for one requirement: the **scanf** function always takes pointers. You never pass a variable value to **scanf**, you always pass the variable address so that **scanf** can store data in the memory location that contains the input variable.

The first argument for **scanf** is always a format string. Additional arguments include the addresses of variables to which values will be assigned.

The program below demonstrates several ways to use **scanf** and various other I/O functions:

```
/* INPUT.C: Read keyboard. */

#include <stdio.h>
#include <conio.h>
#include <ctype.h>
```

```
main()
{
   int num;
   char c;
   char name[80];
   float rb;

   puts( "** Type \"Name:\" and your name" );
   scanf( "Name: %40s", name );
   printf( "** You typed this:\n%s", name );
   puts( "\n\n** Try again, with the gets function." );
   fflush( stdin );
   gets( name );
   printf( "** You typed this:\n%s\n", name );

   printf( "\n** Now type an integer.\n" );
   scanf( "%i", &num );
   sprintf( name, "** You typed this number: %i\n", num );
   puts( name );

   fflush( stdin );
   printf( "** Enter a floating-point value.\n" );
   scanf( "%f", &rb );
   printf( "** The answer is %f or %e\n", rb, rb );

   printf( "** Continue? Y or N\n" );
   do
   {
      c = getch();
      c = tolower( c );
   } while( c != 'y' && c != 'n' );
}
```

First, the **puts** function prints a string that requests input from the user. Then **scanf** reads the input:

```
puts( "** Type \"Name:\" and your name" );
scanf( "Name: %40s", name );
```

Unfortunately, the use of **scanf** for string input creates some difficulties. For one thing, you're forced to type `Name:` before typing the rest of the string. (If you don't type `Name:`, **scanf** won't put a value into the `name` variable.)

A second problem is that **scanf** reads the input stream until it finds a white-space character: a SPACE, TAB, or ENTER.

The prompt below appears on the screen:

```
** Type "Name:" and your name
```

You might type this (you must begin the line with `"Name:"`):

```
Name: F. Scott Fitzgerald
```

The next line takes effect:

```
printf( "** You typed this:\n%s", name );
```

Which prints the following line:

```
** You typed this:
F.
```

The string passed to the **scanf** function told it to expect `"Name:"` and then to read a string, storing it in the `name` variable.

Since the **scanf** function reads strings until it finds a white-space character, the value of `name` is " `F.`" In addition, the words `Scott Fitzgerald` are waiting in the input stream. To clear any stream, use the **fflush** function:

```
puts( "\n\n** Now try it again, with the gets function." );
fflush( stdin );
gets( name );
```

To clear the buffer associated with a stream (including disk files), call **fflush**, passing the pointer to the file or stream. In the example above, `stdin` is the standard input device, the keyboard.

The **puts** function acts like a limited version of **printf**. It prints a string to the standard output device, but can't insert formatted variable values. You pass it a string constant or the name of a string. Also, it always adds a newline to the end of the string it prints.

It is usually preferable to use gets when working with string input.

The **gets** function receives an entire line from the standard input device and places the line in an array of characters. It does not include the newline character typed by the user. It does, however, add a null to the end of the line, to make the series of characters into a string. When you're working with string input, **gets** is generally preferable to **scanf**.

For numeric values, **scanf** is the function of choice:

```
printf( "\n** Now type an integer.\n" );
scanf( "%i", &num );
sprintf( name, "** You typed the number: %i\n", num );
puts( name );
```

The format string **%i** forces **scanf** to treat the input as an integer. The second argument is the address of the variable `num`.

The letter s in **sprintf** marks it as a string function. (There is also a **sscanf** function that handles strings, but we won't discuss it here.) Instead of printing the format string to the screen, as **printf** would do, **sprintf** prints the results to another string. Note that **scanf** requires the address of `num`, but **sprintf** uses its value.

The next **scanf** in program INPUT.C treats the input as a floating-point number:

```
scanf( "%f", &rb );
printf( "** The answer is %f or %e\n", rb, rb );
```

If you enter –555.12, the computer responds:

```
** The answer is -555.119995 or -5.551200e+002
```

Finally, the program uses **getch** to receive a character from the input stream:

```
printf( "** Continue? Y or N\n" );

   do
   {
      c = getch();
      c = tolower( c );
   } while( c != 'y' && c != 'n' );
```

The **getch** function returns a character. That value, in turn, is passed to **tolower**, which converts any uppercase characters to lowercase (in case the CAPS LOCK key is on). Then, the byte is assigned to the variable `c`. The **do** loop continues processing characters until you press `y` or `n`. The program then ends. This simple example ends no matter which key (`y` or `n`) you press. A real program would take some action based on the value returned by the **getch** function.

Standard Disk I/O

If you can read input from the keyboard and write output to the screen, you'll find standard disk files relatively easy to manipulate. There are three rules to remember:

1. You can't do anything with a disk file until you open it. The act of opening a file gives you a **FILE** pointer through which you can access the file.
2. While the file is open, you can use most of the screen and keyboard I/O functions if you precede them with the letter f (**fprintf** instead of **printf**, for example). The file-handling functions work the same as their counterparts, but you must add the **FILE** pointer.
3. When you're finished with a file, it's good programming practice to close it. When **exit** ends the execution of a program, all previously open files are closed (if you'd rather leave them open, use **_exit** instead of **exit**).

Creating and Writing to a Text File

The WRFILE.C program opens a text file, writes a string to it, and closes the file.

```
/* WRFILE.C: Create and write to a disk file. */

#include <stdio.h>

main()
{
   FILE *fp;

   if( (fp = fopen( "c:\\testfile.asc","w" )) != NULL )
   {
      fputs( "Example string", fp );
      fputc( '\n', fp );
      fclose( fp );
   }
   else
      printf( "error message\n" );
}
```

You must include the standard I/O header file (`#include <stdio.h>`) whenever you plan to call input or output functions. It contains essential definitions and prototypes that you need.

The only variable in this program is `fp` which is declared as a pointer to a **FILE**. **FILE** is defined in STDIO.H as a structure of **_iobuf** type, but we don't need to know the specifics. We will refer to the variable `fp` as a "**FILE** pointer."

The first statement combines several operations in one line:

```
if( (fp = fopen( "c:\\testfile.asc", "w" )) != NULL )
```

The **fopen** function opens a file. It expects two parameters, both of which are literal strings or pointers to strings. You provide the name of the file to be opened and the type (read, write, or append). The six types of files are listed in Table 11.3.

Table 11.3 Disk File Types

Type	Action
r	Open an existing file for reading.
w	Create and open a file for writing. Any existing file is replaced. If the file doesn't exist, a new file is created.
a	Open a file for appending. Data is added to the end of an existing file or a new file is created.
r+	Open an existing file for reading and writing.
w+	Create and open a file for reading and writing. An existing file is replaced.
a+	Open a file for reading and appending. Data is added to the end of an existing file or a new file is created.

In WRFILE.C, the file called `c:\testfile.asc` is opened for writing with type `"w"` (a string, not the character `'w'` in single quotes). We plan to write to it.

Notice that the file-name string as it appears in the **fopen** statement contains two backslashes: `"c:\\testfile.asc"`. If you tried to use the string `"c:\testfile. asc"`, which looks correct, the character sequence `\t` would be incorrectly interpreted as a tab character. C automatically converts the two backslashes in the string to a single backslash.

Getting a FILE Pointer

The **fopen** function returns the address of a **FILE**. This value is assigned to `fp`, which is the **FILE** pointer used in all subsequent file operations.

If something goes wrong—if the disk is full or not in the drive or write-protected or whatever—**fopen** doesn't return a **FILE** pointer. When **fopen** fails, it returns a null value.

What we're looking for is any **FILE** pointer that's not null:

```
if( (fp = fopen( "c:\\testfile.asc","w" )) != NULL )
```

Writing to the File

As we saw earlier, **puts** displays a string on the screen. Add an **f** to it and the result is **fputs**, which works similarly. It sends the string to a specified stream (a file) instead of to the standard output device:

```
fputs( "Example string", fp );
```

The function **fputs** takes two parameters: a pointer to the string and the **FILE** pointer. In this and other I/O functions, you refer to the file by name only once (when you use **fopen**). Thereafter, you use its **FILE** pointer.

The **fputs** function writes the entire string to the file but does not include the null that marks the end of the string. Nor does it write a newline character—unless the string already contains a newline.

The **fputc** function writes a character to a file. In the following line, the newline character is sent to the file:

```
fputc( '\n', fp );
```

Closing the File

When the writing is done, **fclose** closes the file:

```
fclose( fp );
```

Conceptually, you can imagine that file I/O functions such as **fputs** and **fputc** write directly to the disk file. In reality, they're storing strings and characters in an intermediate area (called a "memory buffer"). When the buffer fills up, the entire chunk of memory is sent to the file. The process of emptying the buffer is called "flushing." You may forcibly flush the buffer with the **fflush** and **fclose** functions. If you do not close the file before exiting the program, the buffer is not flushed and you may lose data that might remain there.

The **else** clause in WRFILE.C should execute only if something has gone wrong with the **fopen** function:

```
else
   printf( "error message\n" );
```

This line executes if an error occurs when **fopen** tries to create the file. Handling errors is covered in more detail later in this chapter.

Reading a Text File in Binary Mode

The WRFILE.C program that created and wrote to a file was fairly simple. Here's an equally simple program to read the file just created:

```
/* RDFILE.C: Read a file and print characters to the screen. */

#include <stdio.h>

main()
{
   int c;
   FILE *fp;

   if( fp = fopen( "c:\\testfile.asc", "rb" ) )
   {
      while( (c = fgetc( fp )) != EOF )
         printf( " %c\t%d\n", c, c );
      printf( "\nEnd of file marker: %d", c );
      fclose( fp );
   }
   else
      printf( "Error in opening file\n" );
}
```

Although we plan to read the characters as eight-bit entities, the variable `c` should be declared as an **int** instead of a **char**. All of the incoming characters will be bytes the size of a **char**, except one.

When the file has been read from beginning to end, the end-of-file (**EOF**) marker appears on the stream. Within QuickC, an integer value of –1 (0xFFFF) represents **EOF**. To correctly identify this value, the variable `c` must be an integer.

Opening a File for Binary Reading

In the line below, the **fopen** function attempts to open a file. The first argument is the file name; the second is the type and mode, both of which may be literal strings or pointers to strings:

```
if( fp = fopen( "c:\\testfile.asc", "rb" ) )
```

The single backslash character used in path specifications must once again be represented by two backslashes. The file type is **r** for read-only. The additional **b** character forces the file to be read in binary mode instead of text mode. The differences between binary and text files are discussed later in this chapter.

The **fopen** function returns a pointer to a **FILE**. If **fopen** fails, it returns a **NULL** pointer.

Finally, the **if** expression tests for a null value. The original WRFILE.C program included the `!= NULL` test for inequality. Within the test expression of an **if** or a **while**, a 0 value is always false and any other value is considered true. In other words, should `fp` receive a valid nonzero address from **fopen**, the program continues. If something goes wrong, the remaining lines don't execute and the program drops through to the **else**.

Note that the expression above uses an assignment operator (=), not an equality operator (==). The value returned by **fopen** is always assigned to `fp`; they aren't being compared to each other. Then the **if** expression tests that value for truth or falsity.

Getting a Character

The key to the next line in RDFILE.C is the **fgetc** function, to which you pass a **FILE** pointer. It returns the next character from the given file:

```
while( (c = fgetc( fp )) != EOF )
```

The character is assigned to the integer variable `c`. As long as the character doesn't equal **EOF**, the **while** loop continues.

The end-of-file marker equals –1, but it's preferable to use the symbolic constant **EOF**. If the program is transported to another computer, you might find **EOF** has another value. Using the symbolic constant allows you to maintain compatibility between computers and operating systems.

For the same reason, it's preferable to test for **NULL** instead of assuming that **NULL** will always equal 0.

Viewing the File

Since there's only one line inside the **while** loop, it's not necessary to enclose it in curly braces. The variable `c` contains the character read from the file. It then can be printed:

```
printf( " %c\t%d\n", c, c );
```

The characters from the file print twice, once as a character (**%c**) and once as a decimal number (**%d**), separated by a tab stop. This **printf** statement repeats until **fgetc** (inside the **while** loop) finds no more characters in the file.

Binary and Text Files

Normally, you wouldn't write a file in text mode and then read it in binary mode. As a general rule, you pick whichever mode is more appropriate (text mode for text or binary mode for data) and stick with it.

A somewhat baffling thing happened in the example above, however. The WRFILE.C program wrote `"Example string"` to a disk file and then added a newline character. That should be a total of 15 characters. But if you examine the directory, you'll see the file uses 16 bytes.

Where did the extra byte come from?

Testing Text Mode

If you ran the RDFILE.C program, you probably noticed two characters followed the line: a carriage return (ASCII 13) and a linefeed (ASCII 10). If you make the following change to the program, the output of RDFILE.C is different:

```
if( (fp = fopen( "c:\\testfile.asc","rt" )) != NULL )
```

The only modification is that the second string is `"rt"` instead of `"rb"`. The **t** represents text mode; the **b** is binary mode. If you don't specify a mode, the **fopen** function defaults to text mode.

The list below shows the output of the two programs.

RDFILE.C (binary mode)	RDFILE.C (text mode)
E 69	E 69
x 120	x 120
a 97	a 97
m 109	m 109
p 112	p 112
l 108	l 108
e 101	e 101
32	32
s 115	s 115
t 116	t 116
r 114	r 114
i 105	i 105
n 110	n 110
g 103	g 103
13	10
10	End-of-file marker: -1
End-of-file marker: -1	

In binary mode there seems to be two characters after the string. In text mode there's only one.

Ends-of-Lines, Ends-of-Files

The two modes—binary and text—treat end-of-line (**EOL**) characters and end-of-file (**EOF**) characters in different ways.

In DOS, a line of text ends with a carriage return (CR) and a linefeed (LF), which appear above as ASCII 13 plus ASCII 10. In the UNIX operating system, which has close ties to the C language, a single ASCII 10 (the newline character) marks the end of a line.

The once-popular CP/M operating system signals the end of files with a CTRL+Z character (ASCII 26, 0x1A)—a tradition that carried forward to DOS. This is not the case with UNIX (and C), which don't use a unique **EOF** character.

Text Mode Translations

It's important to understand the differences between text mode and binary mode when writing and reading disk files. No translations are made in binary mode. In text mode, however, the end-of-line and end-of-file characters are translated.

When you read a file in text mode and a CR-LF combination appears in the stream, the two characters are translated to one newline character. The opposite translation occurs when you write a file in text mode: each CR-LF combination is translated to one newline character. In other words, the newline is represented by two characters on disk and one character in memory. These translations do not occur when you read and write a file in binary mode.

When you read a file in text mode and a CTRL+Z (0x1A) character appears in the stream, the character is interpreted as the end-of-file character. However, when you're in text mode and you close a file to which you've been writing, a CTRL+Z is not placed in the file as the last character. In binary mode, the CTRL+Z character has no special meaning (it is not interpreted as the end-of-file character).

The difference between text mode and binary mode is relatively minor when you're handling strings, but it's important when you're writing numeric values to disk files.

Text Format for Numeric Variables

Many programs, of course, use numeric as well as character data. When you wish to save numbers, you have two choices: text mode or binary mode. The SVTEXT.C program below illustrates the less desirable way of creating files for numeric variables.

```
/* SVTEXT.C: Save integer variables as text. */

#include <stdio.h>

int list[] = { 53, -23456, 50, 500, 5000, -99 };
extern int errno;
char fname[] = "numtext";
char temp[81];

main()
{
   FILE *fptr;
   int i;

   if( (fptr = fopen( "numtext","wt" )) != NULL )
   {
      for( i=0; i<6; i++ )
         fprintf( fptr, "Item %d: %6d \n", i, list[i] );
      fclose( fptr );
   }
   else
      printf( "Error: Couldn't create file.\n" );

   if( (fptr = fopen( "badname", "rt" )) != NULL )
   {
      /* do nothing */
   }
   else
   {
      printf( "Error number: %d\n\t", errno );
      perror( "Couldn't open file BADNAME\n\t" );
   }

   if( (fptr = fopen( fname, "rt" )) != NULL )
   {
      list[0] = 0;
      fscanf( fptr, "Item %d: %d \n", &i, &list[0] );
      printf( "Values read from file:\t %d %d\n", i, list[0] );
      fgets( temp, 80, fptr );
      printf( "String from file: \t%s\n", temp );
      while( (i = fgetc( fptr )) != '\n' )
         printf( "char: %c \t ASCII: %d \n", i, i );
      rewind( fptr );
      printf( "Rewind to start -->\t%s", fgets( temp, 80, fptr ) );
      fclose( fptr );
   }
   else      printf( "Trouble opening %s \n", fname );
}
```

The SVTEXT.C program does three things:

1. First, it creates a text file called NUMTEXT. If you TYPE NUMTEXT from the DOS prompt or load the NUMTEXT file into a word processor, it looks like this:

```
Item 0:     53
Item 1: -23456
Item 2:     50
Item 3:    500
Item 4:   5000
Item 5:    -99
```

2. Next, SVTEXT.C deliberately attempts to open a nonexistent file called BADNAME, to cause a disk error. This section serves no purpose except to illustrate error handling.

3. Finally, it reads parts of NUMTEXT, using several file-input functions.

Opening the File for Writing

By now, the **fopen** function should look familiar to you. The only change in the block below is the `"wt"` mode. The **fopen** function returns a **NULL** if any errors occur, so the block after the **if** should execute if **fopen** succeeds.

```
if( (fptr = fopen( "numtext","wt" )) != NULL )
{
   for( i=0; i<6; i++ )
      fprintf( fptr, "Item %d: %6d \n", i, list[i] );
   fclose( fptr );
}
else
   printf( "Error: Couldn't create file.\n" );
```

The **for** loop counts from 0 to 5, printing 6 strings to the file. The **fprintf** function works the same as **printf** with one change. You must place the **FILE** pointer before the format string.

Error Handling

To illustrate what happens when something goes wrong, the next line creates a disk error (as long as you don't have a file called BADNAME in your working directory).

```
if( (fptr = fopen( "badname", "rt" )) != NULL )
```

The **if** block is empty, because we expect the program to drop through to the **else** clause that handles errors:

```
else
{
   printf( "Error number: %d\n\t", errno );
   perror( "Couldn't open file BADNAME\n\t" );
}
```

The **else** block shows two ways you can deal with errors. Note that the **errno** variable, which was declared as an external integer, has never been assigned a value. QuickC automatically puts error numbers into **errno**. In this program, the error number is printed to the screen. In your own programs, you might wish to branch to various error-handling routines, based on the value in the system variable **errno**. For a list of values for **errno**, see the individual online help entries for file-handling functions.

It's important to remember that the standard output device is the screen and that **printf** sends messages to **stdout**. However, if you redirect output to a disk file, using a command line such as `SVTEXT > MYFILE`, the **printf** statement prints the error message to MYFILE. In most cases, you'd prefer to see the error message on the screen.

The second, and better, way to handle I/O errors is the **perror** function, which prints two strings: one that you pass to it and one that spells out—in English—the error message. This message goes to the standard error stream (**stderr**), which is always the screen, regardless of whether you've redirected output or not. For this reason, **perror** is preferable to **printf** for printing error messages.

The error messages should look like this on your screen:

```
Error number: 2
   Couldn't open file BADNAME
   : No such file or directory
```

Reading Text with fscanf

The final **fopen** in SVTEXT.C opens the file created earlier:

```
if( (fptr = fopen( fname, "rt" )) != NULL )
```

Note that we passed the name of a string rather than a literal string.

Below, **fscanf** reads in two numeric variables from the first string in the file. Note that it works the same as **scanf**, but you add the **FILE** pointer as the first argument:

```
fscanf( fptr, "Item %d: %d \n", &i, &list[0] );
printf( "Values read from file:\t %d %d\n", i, list[0] );
```

Reading Text with fgets and fgetc

At this point, the first line in the file has been read and converted to two integer values. The file is straight text, so you can treat the second line as a string:

```
fgets( temp, 80, fptr );
printf( "String from file: \t%s\n", temp );
```

The **fgets** function requires three arguments: a pointer to a string, the maximum number of characters to read, and the **FILE** pointer. The function stops reading characters when it encounters a newline character or when it reaches the maximum number of characters or the end of the file.

If you prefer, you can input the characters one by one:

```
while( (i = fgetc( fptr )) != '\n' )
   printf( "char: %c \t ASCII: %d \n", i, i );
```

The **printf** inside the **while** loop prints each character as a character (**%c**) and also as a decimal value (**%d**). The **while** loop continues reading characters until it finds the end of the line.

Back to the Beginning

The **rewind** function resets the position pointer to the beginning of the file. In the program line below, the first line from the file is printed:

```
rewind( fptr );
printf( "Rewind to start -->\t%s", fgets( temp, 80, fptr ) );
```

The screen output looks like this:

```
Error number: 2
        Couldn't open file BADNAME
        : No such file or directory
Values read from file:        0 53
String from file:        Item 1: -23456

char: I         ASCII: 73
char: t         ASCII: 116
char: e         ASCII: 101
char: m         ASCII: 109
char:           ASCII: 32
char: 2         ASCII: 50
char: :         ASCII: 58
char:           ASCII: 32
char:           ASCII: 32
char:           ASCII: 32
char:           ASCII: 32
char:           ASCII: 32
char: 5         ASCII: 53
char: 0         ASCII: 48
char:           ASCII: 32
Rewind to start -->        Item 0:     53
```

It is inefficient to store numeric data in text format.

There seem to be quite a few white-space characters in the text file. Text files are great for text, but they store numeric values in a wasteful way. Binary format offers several advantages.

Using Binary Format

When you're processing strings of ASCII characters and writing them to disk files, it matters little whether you use text mode or binary mode, as long as you're consistent. The advantage of text mode is that it translates newlines to the carriage-return–line-feed combination, making it possible to use the DOS TYPE command to view the file.

When you're processing numeric values (integers and floating-point numbers), however, you may wish to save your variables in binary mode files, in binary format, for the following reasons:

- Binary format almost always saves disk space. In text mode, the number 12345.678 would require eight bytes for the ASCII numerals, one byte for the decimal point, and one or more bytes for a separator between variables. In binary format, a floating-point number uses four bytes, regardless of its value. Short integers use only two bytes.
- Binary format generally saves computer time. When you use **fprintf** to print a numeric value to disk, the computer must translate the internal binary representation to a series of characters. Likewise, when **fscanf** reads characters into memory, the ASCII values must be translated to the internal binary format. In binary format, none of these translations takes place.
- Binary format preserves the precision of floating-point numbers. The translation from binary to decimal ASCII and back to binary affects the precision of the value.
- A binary save of arrays or structures is fast. It's not necessary to read through an array of 100 items and print each one to the disk file. Instead, you call the **fwrite** function (discussed below) once, passing it the size of the array to be saved.

NOTE *Binary mode is separate from binary format. The modes (binary and text) are parameters you pass to the* ***fopen*** *function. They affect the translation of newlines and the placing of* ***EOF*** *markers. The formats (binary and text) are ways of representing numeric values. An integer in binary format always occupies two bytes on disk. An integer in text format uses a variable number of bytes: it might contain one character (5) or six (–10186).*

Opening a Binary File

The SVBIN.C program below creates two binary mode files with the variables saved in binary format:

```
/* SVBIN.C: Save integer variables in binary format. */

#include <stdio.h>
#define ASIZE 10

main()
{
   FILE *ap;
   int zebra[ASIZE], acopy[ASIZE], bcopy[ASIZE];
   int i;

   for( i = 0; i < ASIZE; i++ )
      zebra[i] = 7700 + i;

   if( (ap = fopen( "binfile", "wb" )) != NULL )
   {
      fwrite( zebra, sizeof(zebra), 1, ap );
      fclose( ap );
   }
   else
      perror( "Write error" );

   if( (ap = fopen( "morebin", "wb" )) != NULL )
   {
      fwrite( &zebra[0], sizeof(zebra[0]), ASIZE, ap );
      fclose( ap );
   }
   else
      perror( "Write error" );

   if( (ap = fopen( "binfile", "rb" )) != NULL )
   {
      printf( "Hexadecimal values in binfile:\n" );
      while( (i = fgetc( ap )) != EOF )
         printf( "%02X ", i );
      rewind( ap );
      fread( acopy, sizeof(acopy), 1, ap );
      rewind( ap );
      fread( &bcopy[0], sizeof( bcopy[0] ), ASIZE, ap);
      for( i=0; i<ASIZE; i++ )
         printf( "\nItem %d = %d\t%d", i, acopy[i], bcopy[i] );
      fclose( ap );

   }
   else
      perror( "Read error" );

}
```

Focus your attention on the `zebra` array. It contains 10 integers, because the array size `ASIZE` was defined as 10. First, some values are stored in `zebra` (in a moment, we'll see why 7700–7709 are significant):

```
for( i = 0; i < ASIZE; i++ )
   zebra[i] = 7700 + i;
```

Next, we open a file and use **fwrite** to write the entire array to disk:

```
if( (ap = fopen( "binfile", "wb" )) != NULL )
{
   fwrite( zebra, sizeof(zebra), 1, ap );
   fclose( ap );
}
```

Writing an Array in One Line

The **fwrite** function requires four pieces of information:

1. The address of the item (a variable, array, or structure)
2. The size of the item in bytes
3. The number of items to be written
4. The **FILE** pointer for a previously opened binary mode file

In this example, the first argument, `zebra` is an array and, as you may remember from Chapter 8, "Pointers," the name of an array is the address of the array.

To provide the second argument for **fwrite**, SVBIN.C uses the **sizeof** operator, which returns the number of bytes a variable requires. Because `zebra` is an array of 10 integers and integers use 2 bytes each, the size of `zebra` should be 20. If you view a directory of your disk after running this program, you'll notice that the file BINFILE is exactly 20 bytes long.

The third argument tells **fwrite** how many items to write to the file. We have 1 array, so this parameter is 1.

The fourth argument is the **FILE** pointer returned by **fopen**.

There's another way to copy the 20 bytes of `zebra` to the file. After writing to BINFILE, the program uses the **fopen** function to create a second file called MOREBIN. The following **fwrite** line writes 10 integers instead of 1 array:

```
fwrite( &zebra[0], sizeof(zebra[0]), ASIZE, ap );
```

The second and third arguments have changed. Instead of passing the size of the array (20) and writing 1 copy of the array, we're accessing the size of 1 element (2 bytes) and writing 10 of them (using the symbolic constant **ASIZE**). The contents of this disk file should match, byte for byte, the contents of BINFILE.

Examining the Binary Contents

Finally, we look at what's inside the file BINFILE. It is opened for reading as a binary file:

```
if( (ap = fopen( "binfile", "rb" )) != NULL )
```

A short **while** loop reads the bytes from BINFILE and displays them in hexadecimal notation:

```
printf( "Hexadecimal values in binfile:\n" );
while( (i = fgetc( ap )) != EOF )
   printf( "%02X ", i );
```

After running SVBIN.C, the screen displays these values:

```
14 1E 15 1E 16 1E 17 1E 18 1E
19 1E 1A 1E 1B 1E 1C 1E 1D 1E
```

The low byte precedes the high byte, so the first two bytes represent the number 0x1E14, which is 7700 in decimal. The next two bytes equal 7701, and so on.

A curious thing happens when you run SVBIN.C and then try to treat the 20-byte file as text. If you TYPE BINFILE from the DOS command line, the file appears as gibberish (of course), and you see only 12 of the 20 characters on the screen. Where did the other characters go? Recall the previous discussion of binary and text files. In DOS, a CTRL+Z (0x1A) marks the end of a text file. And in the midst of our binary file is one of those **EOF** characters. It's not acting as an **EOF**; it's part of the number 0x1E1A. But if you ever open this file in text mode, you'll be unable to read past the twelfth byte.

Retrieving the Values from Disk

Most of the time, you won't want to read a binary file one byte at a time. Instead, you call **fread**, which reads a disk file and stores the values in a variable, an array, or a structure. The **fread** function complements **fwrite**. It takes four parameters:

1. The address of the variable
2. The size of the variable in bytes
3. The number of values to read
4. The **FILE** pointer that references a binary file opened for reading

Here's one way to read values into an array:

```
rewind( ap );
fread( acopy, sizeof( acopy ), 1, ap );
```

The **rewind** command is necessary because we've already read through the file once. The `acopy` and `bcopy` arrays are the same size as our original `zebra` array. To fill an array with this technique, pass the address, the size of the entire array, a number 1, and the **FILE** pointer.

A second way to fill an array is to pass the size of a single element and the number of elements you want to read:

```
rewind( ap );
fread( &bcopy[0], sizeof( bcopy[0] ), ASIZE, ap );
```

In the first example of **fread**, we pass the information that the array `acopy` is 20 bytes long and we want to read it once. In the second example, we pass the size of an integer (2 bytes) and ask for 10 of them. In either case, 20 bytes are transferred.

Just to make sure both arrays are equal, we can print them out:

```
for( i = 0; i < ASIZE; i++ )
printf( "\nItem %d = %d\t%d", i, acopy[i], bcopy[i] );
fclose( ap );
```

The screen displays the values 7700 through 7709, which survived the trek from `zebra` to BINFILE and back again. These values were stored in the `zebra` array, written to a binary file, then read back into the `acopy` and `bcopy` arrays.

Low-Level Input and Output

The file-handling routines such as **fopen**, **fprintf**, and **fclose** are called "standard" because they're defined in the ANSI standard. Code that uses the standard routines will generally be portable from one machine to another.

In addition to the standard file-handling functions, the QuickC library includes some low-level I/O functions, which allow more direct access to disk files.

The low-level I/O routines (also called "system-level") are generally not portable. They work in DOS and OS/2, but they may not work elsewhere. They're also a little more difficult to use. Instead of declaring a pointer to a FILE structure, you must allocate your own buffer and manage transfer of the bytes yourself. You move values into the buffer, then send the contents of the buffer to disk.

Low-level routines can be more efficient, but they usually aren't portable.

Low-level routines have some advantages, though. One is that you have more control over the machine. Another is that low-level I/O can be faster than standard I/O, if you know what you're doing. The choice is up to you: portability versus efficiency. You should choose one or the other; it's not a good idea to mix standard and system-level routines.

Low-Level Reading and Writing

The program RWFILE.C illustrates some of the low-level, file-handling commands. It creates a file, writes to it, and closes it. Then the file is opened for reading and the contents of the file are displayed on the screen.

```
/* RWFILE.C: Read and write a file. */

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>

#define BUFF 512

main()
{
   char inbuffer[BUFF];
   char outbuffer[BUFF];
   int infile, outfile, length, num;

   strcpy( outbuffer, "Happy Birthday." );
   length = strlen( outbuffer );
   length++;

   if( (outfile = open( "testfile.bin",
      O_CREAT | O_WRONLY | O_BINARY,  S_IWRITE )) != -1 )
   {
      if( (num = write( outfile, outbuffer, length )) == -1 )
         perror( "Error in writing" );
      printf( "\nBytes written to file: %d\n", num );
      close( outfile );
   }
   else
      perror( "Error opening outfile" );
```

```
    if( (infile = open( "testfile.bin", O_RDONLY | O_BINARY )) != -1  )
    {
       while( length = read( infile, inbuffer, BUFF ) )
          printf( "%d bytes received so far.\n", length );
       close( infile );
       printf( "%s\n", inbuffer );
    }
    else
       perror( "Error opening infile" );
}
```

Several header files must be included:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
```

The symbolic constant `BUFF` is defined as 512. This value is used immediately in the declaration of two buffers:

```
char inbuffer[BUFF];
char outbuffer[BUFF];
```

Note that we don't need **FILE** structures anywhere in the program. The standard I/O routines automatically allocated space for a buffer. Since we're operating closer to the DOS level, we must allocate our own buffers, instead of relying on the system. If you set the buffer size to a sufficiently large value, QuickC will run out of stack space. When this happens, you may either make the buffers global variables or use the **malloc** function to allocate an additional chunk of memory.

The **open** function takes three parameters:

```
if( (outfile = open( "testfile.bin",
   O_CREAT | O_WRONLY | O_BINARY, S_IWRITE )) != -1 )
```

The first parameter is the file name. The second is a sequence of "oflags" that are combined with the OR operator. The oflags determine which type of file will be opened: it will be created (`O_CREAT`), it will be write-only (`O_WRONLY`), and it will be a binary—not a text—file (`O_BINARY`). When you create a new file, you must include the third parameter: `S_IWRITE`.

The **open** function returns a file handle, which is assigned to the integer variable `outfile`. Note that this is an integer, not a pointer to a **FILE** structure. If anything goes wrong, a value of –1 is returned by **open**, and we should test for this.

Table 11.4 summarizes the differences between **fopen** and **open**.

Table 11.4 Standard vs. Low-Level

Function	Parameters	Returns	Error Condition
fopen	File name, type (**r**, **w**, **a**), and mode (**t**, **b**)	Pointer to **FILE**	NULL
open	File name, oflags	File handle (integer)	–1

Low-Level Writing

The **write** function takes three parameters:

1. The file handle returned by **open**
2. The address of the buffer
3. The number of bytes to write

You, the programmer, are responsible for filling up the buffer. The **write** function returns the number of bytes actually written to the file.

```
if( (num = write( outfile, outbuffer, length )) == -1 )
   perror( "Error in writing" );
printf( "\nBytes written to file: %d\n", num );
close( outfile );
```

Low-Level Reading

Next, we open the file for reading. Again, the oflags are required:

```
if( (infile = open( "testfile.bin", O_RDONLY | O_BINARY )) != -1 )
{
   while( length = read( infile, inbuffer, BUFF ) )
      printf( "%d bytes received so far.\n", length );
   close( infile );
   printf( "%s\n", inbuffer );
}
```

The **read** function takes three parameters:

1. The file handle
2. The address of the buffer
3. The size of the buffer

The value returned is the number of bytes read. The **while** loop continues as long as there are characters in the stream. In a real application, you'll have to handle the bytes stored in the buffer.

The low-level file functions are unbuffered. When you call **write**, the bytes are written directly to the disk file. The standard file function **fwrite** doesn't write data to disk; it writes to a buffer. The buffer is transferred to disk when the buffer fills up, when the **fflush** function is called, or when the file is closed. As a general rule, you should not mix buffered and unbuffered routines. Use the standard routines or the low-level routines, but not both.

This chapter started with keyboard input and screen output and led into discussions of file I/O. The following chapters cover in greater depth assembly language routines and some specialized types of screen output, including high-resolution graphics, fonts, and presentation graphics.

CHAPTER

12

Dynamic Memory Allocation

A program that allocates memory "dynamically" (as it runs) can respond flexibly to a user's needs, creating new data structures when the need arises and discarding them when their job is done.

As you read this chapter you'll learn how to allocate memory with the **malloc** library function and free memory with the **free** function. We'll also look at two related functions, **calloc** and **realloc**.

Memory allocation requires the use of pointers. If you're not familiar with pointers, read Chapter 8, "Pointers," before tackling this chapter.

Why Allocate?

The malloc family of library functions can allocate memory during run time.

The **malloc** (memory allocate) family of library functions enables you to allocate blocks of memory dynamically. The capability to create new data structures on the fly lets you tailor a program's behavior precisely to the user's needs.

For simple programs, such as the examples in Part 1, memory allocation is largely automatic. When you declare variables, as in the lines

```
int count;
char buffer[160];
```

QuickC allocates enough memory to store each variable (2 bytes for the first variable and 160 for the second). This method works fine if you know each variable's size in advance. Some program memory needs aren't easy to predict, however.

To take a simple example, say you write an address-book program that stores addresses in an array of structures. A novice programmer might begin by declaring an array of, say, 100 structures, in the following manner,

```
struct address list[100];
```

but this approach is needlessly limiting. If your list contains only a few addresses, most of the memory in the array is wasted. And if you want to enter more than 100 addresses, you're out of luck.

A better approach is to allocate memory for the array dynamically. This way, the program can use only as much memory as needed for the current address list. Each time you add an address, or delete one, the program can expand or shrink the array as needed.

Memory Allocation Basics

We'll use a simple example program, COPYFILE.C, to demonstrate the basics of dynamic memory allocation—how to allocate a memory block, access its contents, and free the block when its purpose is served.

The COPYFILE.C program, shown below, dynamically allocates a buffer that it uses to store file data.

```
/* COPYFILE.C: Demonstrate malloc and free functions. */

#include <stdio.h>     /* printf function and NULL */
#include <io.h>        /* low-level I/O functions */
#include <conio.h>     /* getch function */
#include <sys\types.h> /* struct members used in stat.h */
#include <sys\stat.h>  /* S_ constants */
#include <fcntl.h>     /* O_ constants */
#include <malloc.h>    /* malloc function */
#include <errno.h>     /* errno global variable */

int copyfile( char *source, char *destin );

main( int argc, char *argv[] )
{
   if( argc == 3 )
      if( copyfile( argv[1], argv[2] ) )
         printf( "Copy failed\n" );
      else
         printf( "Copy successful\n" );
   else
      printf( "  SYNTAX: COPYFILE <source> <target>\n" );

   return 0;
}

int copyfile( char *source, char *target )
{
   char *buf;
   int hsource, htarget, ch;
   unsigned count = 50000;
```

```
    if( (hsource = open( source, O_BINARY | O_RDONLY )) == - 1 )
        return errno;
    htarget = open( target, O_BINARY | O_WRONLY | O_CREAT | O_EXCL,
                            S_IREAD | S_IWRITE );
    if( errno == EEXIST )
    {
        cputs( "Target exists. Overwrite? " );
        ch = getch();
        if( (ch == 'y') || (ch == 'Y') )
            htarget = open( target, O_BINARY | O_WRONLY | O_CREAT | O_TRUNC,
                                    S_IREAD | S_IWRITE );
        printf( "\n" );
    }
    if( htarget == -1 )
        return errno;

    if( filelength( hsource ) < count )
        count = (int)filelength( hsource );

    buf = (char *)malloc( (size_t)count );

    if( buf == NULL )
    {
        count = _memmax();
        buf = (char *)malloc( (size_t)count );
        if( buf == NULL )
            return ENOMEM;
    }

    while( !eof( hsource ) )
    {
        if( (count = read( hsource, buf, count )) == -1 )
            return errno;
        if( (count = write( htarget, buf, count )) == - 1 )
            return errno;
    }

    close( hsource );
    close( htarget );
    free( buf );
    return 0;
}
```

Before we look at how COPYFILE.C works, let's note what it does. Unlike the DOS COPY command, the COPYFILE.C program asks for confirmation before overwriting an existing file. The program expects to receive two file names as command-line parameters: the name of the file to copy and the name of the new file. For instance, the following command copies the file SAMPLE.EXE to the new file EXAMPLE.EXE:

```
copyfile sample.exe example.exe
```

If the target file already exists, COPYFILE.C displays:

```
Target exists. Overwrite?
```

COPYFILE.C overwrites an existing file only if the user presses the Y key in response.

Preparing to Allocate Memory

The COPYFILE.C program copies the source file in chunks, using an allocated memory block as a buffer for file data. The following program lines are the ones involved in allocating and freeing the memory block. (These are taken from the program in order, but are not consecutive.)

```
#include <malloc.h>    /* malloc function */
char *buf;
unsigned count = 50000;
buf = (char *)malloc( (size_t)count );
free( buf );
```

The first of these,

```
#include <malloc.h>  /* malloc function */
```

includes the standard include file MALLOC.H, which contains declarations for **malloc** and other memory-allocating functions.

The **malloc** function, which the program will call to allocate a memory block, returns the address where the block begins. COPYFILE.C declares the pointer variable `buf` to store this address:

```
char *buf;
```

As you'll see shortly, the pointer `buf` will be initialized to point to the allocated block. Once this is done, the program can access the block's contents through the pointer.

The COPYFILE.C program declares another variable, `count`, which is used to tell **malloc** how much memory (in bytes) to allocate. The program initially sets this value to 50,000:

```
unsigned count = 50000;
```

If the source file is smaller than 50,000 bytes, COPYFILE.C later resets `count` to the smaller value.

Specifying the Size of the Allocated Block

Now we're ready to allocate the block. The statement

```
buf = (char *)malloc( (size_t)count) );
```

in COPYFILE.C calls the **malloc** function, passing the value of `count` as an argument. This argument indicates the size of the desired block in bytes. In COPYFILE.C this value is 50,000 or the size of the source file, whichever is smaller.

Look at the type cast preceding the function argument:

```
(size_t)
```

The cast is performed for ANSI compatibility (**malloc** is part of the ANSI standard). Under ANSI, **malloc** is declared as taking an argument of the type `size_t`. To ensure the portability of your programs, the value passed to **malloc** should be either declared or cast as type `size_t`.

A Graphic Illustration

Figures 12.1 and 12.2 show how the COPYFILE.C program allocates a memory block. The figures are simplified and are not drawn to scale. They represent the program's "data segment," which is the memory area available for the program's data storage.

NOTE *The details of data storage differ depending on the current "memory model," an advanced concept that goes beyond the scope of this book. For purposes of discussion, this book assumes the small memory model, which is the default for QuickC.*

Figure 12.1 represents the program's data segment before COPYFILE.C allocates a block of memory. The shaded area labeled "Declared data" contains the program's declared variables and "stack," which is used for temporary storage. The unshaded area labeled "Heap" contains the memory available for allocation by COPYFILE.C.

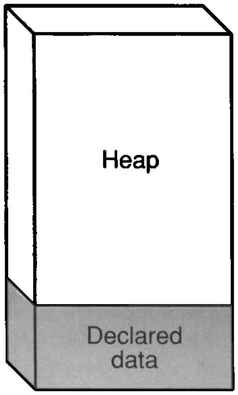


Figure 12.1 Before Allocating a Memory Block

Figure 12.2 shows COPYFILE.C immediately after the program calls the **malloc** function to allocate a block of memory. The allocated block is taken from heap memory and lies directly above the program. If COPYFILE.C allocated a second memory block, that block would lie above the first, further diminishing heap memory.

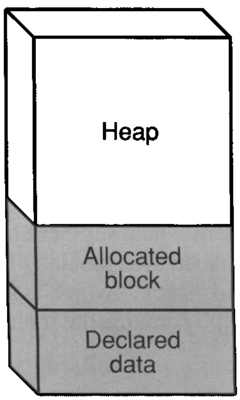


Figure 12.2 After Allocating a Memory Block

While it's common to say the **malloc** function "creates" a memory block, that terminology is a bit misleading. As Figures 12.1 and 12.2 show, **malloc** simply gives your program control over memory that's already present.

The malloc function does not initialize the memory it allocates.

Since the allocated block has been present in memory all along, it may contain random values or values left over from some previous use. The **malloc** function doesn't initialize allocated memory. Substitute the **calloc** function for **malloc** if

you want to clear an allocated block before use. (See the section "The calloc Function" below.)

Assigning the Address that malloc Returns

If the call to **malloc** succeeds, **malloc** returns the address of the memory block it allocates. COPYFILE.C assigns that return value to the pointer variable `buf` and then accesses the allocated block through `buf`.

Before assigning the address that **malloc** returns, COPYFILE.C performs a type cast on the address

```
(char *)
```

The type cast indicates which type of memory you are allocating. Prior to the ANSI standard, **malloc** was declared as returning a pointer to type **char**, so it was necessary to cast the return value when assigning the value to any other pointer type.

Under ANSI, **malloc** returns a pointer to type **void**. Since a **void** pointer can be converted to any pointer type, it's not strictly necessary to cast the return from **malloc**. (If you omit the cast, QuickC does a silent type conversion.) The type cast improves readability, however.

Checking the Return from malloc

If a call to **malloc** fails—usually because not enough memory is available—the function returns a null pointer (defined as `NULL` in the standard include file STDIO.H). You should always test this return value, even if you're confident the allocation will succeed. If you ignore the return value and access memory through a null pointer, your program may stop with a run-time error or overwrite unpredictable memory addresses.

Thus, before attempting to use the allocated memory block, COPYFILE.C checks to make sure the call to **malloc** succeeded:

```
if( buf == NULL )
{
.
.
.
}
```

The **if** statement tests whether the pointer `buf` has been set to `NULL`, which would signal failure. In that case, the program executes the code within the braces of the **if** statement.

Sometimes there may be enough free memory to satisfy only part of your memory request. Look at how COPYFILE.C handles this situation:

```
buf = (char *)malloc( (size_t)count );

if( buf == NULL )
{
   count = _memmax();
   buf = (char *)malloc( (size_t)count );
   if( buf == NULL )
      return ENOMEM;
}
```

If fewer than `count` bytes of memory are available, the initial call to **malloc** returns `NULL`, indicating failure. In that event, COPYFILE.C calls the **_memmax** library routine to find how much memory is available and assigns that value to the variable `count`:

```
count = _memmax();
```

Then COPYFILE.C calls **malloc** again, requesting a smaller amount of memory. This request is bound to succeed unless no memory is available.

Accessing an Allocated Memory Block

Once you have allocated a block of memory, you can access it through its pointer (`buf`, in this example). COPYFILE.C uses its allocated block as a file buffer, alternately reading in data from the source file, through the statement

```
if( (count = read( hsource, buf, count )) == -1 )
   return errno;
```

and writing it to the target file, through the statement

```
if( (count = write( htarget, buf, count )) == - 1 )
   return errno;
```

The **read** and **write** function calls occur within **if** statements that compare the function return values to –1, which would indicate failure.

COPYFILE.C treats its allocated block as a single chunk of memory. To access individual data items in an allocated block, you can use either pointer or array notation. Both of the following statements, for instance, refer to the third byte in the block that `buf` points to:

```
buf[2] = 'x';
*(buf+2) = 'x';
```

Allocating Memory for Different Data Types

Since COPYFILE.C accesses its allocated block through a **char** pointer, the program must treat the items in that block as **char** types. If you need to use a different type of memory, simply change the pointer declaration and cast the return from **malloc** accordingly. For instance, you could use the following statements to allocate a block large enough to store 30 **int** values:

```
int *buf;

buf = (int *)malloc( (size_t)sizeof( int ) * 30 );
```

Here, the **sizeof** operator eliminates the need to calculate how many bytes of storage 30 integers require. The expression

```
sizeof( int )
```

returns the size of an **int** type, which we then multiply by the desired number of **int** items.

If the above call to **malloc** succeeds, you have, in effect, a 30-element array of integers. And since pointer notation and array notation are interchangeable, you can access any element of the array using the pointer name and array notation. For instance, the expression

```
ptr[2] = 50;
```

assigns the value 50 to the third element of the array. Note that this statement accesses the third **int** element in the array, not the third byte. Pointer references, as explained in Chapter 8, "Pointers," are always scaled by the size of the type used to declare the pointer.

Allocating memory for structures is equally straightforward. Say that you want to allocate memory to store 10 structures of the type `employee`, which is declared in the EMPLOYEE.C program in Chapter 4, "Data Types." The EMPLOYEE.C program uses the following structure type:

```
struct employee
{
   char name[10];
   int months;
   float wage;
};
```

You could use the statement

```
struct employee *e_ptr;
```

to declare a pointer to an item of the `employee` type. Once you have a suitable pointer, you could use the following statement to allocate enough memory to store 10 structures of the same type:

```
e_ptr = (struct employee *) malloc( (size_t)sizeof( struct employee ) * 10 );
```

Here, the **sizeof** operator

```
sizeof( struct employee )
```

returns the size of a structure of the `employee` type.

If the allocation succeeds, you have, in effect, an array of structures of type `employee`. Using structure notation, you can access any structure member in the block. The following statements, for instance, initialize the members of the third structure in the array:

```
strcpy( e_ptr[2].name, "Isaac, N." );
e_ptr[2].months = 54;
e_ptr[2].wage = (float) 12.21;
```

Deallocating Memory with the free Function

The free function deallocates an allocated memory block.

When you have finished using an allocated memory block, you should free (deallocate) the block with the **free** library function. The **free** function takes one argument: the address of the block you wish to free. The COPYFILE.C program frees its allocated block with the statement:

```
free( buf );
```

It's your responsibility to pass a valid address to **free**. Unlike most library functions, **free** doesn't return any value to indicate success or failure. If you pass an invalid address, the memory block remains allocated and can't be used for any other purpose.

Figure 12.3 shows COPYFILE.C after the program frees its allocated block. The **free** function releases the block from the program's control, returning it to the heap. The same memory is still present, of course. But since your program no longer has control of that memory, you shouldn't attempt to use it. (See "Using Dangling Pointers" in Chapter 10, "Programming Pitfalls," for more information on this point.)

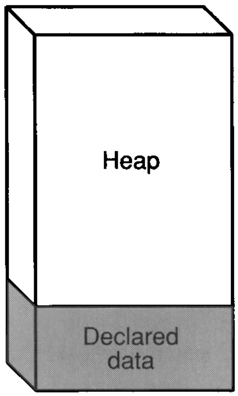


Figure 12.3 After Freeing the Allocated Memory Block

Specialized Memory-Allocating Functions

The C library contains two specialized versions of **malloc** that you may find useful. The **calloc** function allocates memory for an array and sets the block's contents to 0. The **realloc** function can expand or shrink an existing memory block.

The calloc Function

The **calloc** (calculated allocate) function is especially useful for allocating memory for an array. It works like **malloc** but takes two arguments:

- The number of data items for which you wish to allocate memory
- The size of each data item

This scheme eliminates the need for you to calculate the number of bytes needed to store the desired array. For instance, the statement

```
ptr = (int *) calloc( (size_t)30, (size_t)sizeof( int ) );
```

allocates enough memory for a 30-element integer array, and

```
e_ptr = (struct employee *) calloc( (size_t)30, sizeof( struct employee ) );
```

allocates enough memory for a 30-element array of structures of type `employee`.

The calloc library function allocates memory and sets every byte in the block to 0.

The **calloc** function also sets every byte in the requested block to 0. The **malloc** function, as we noted earlier, doesn't change the contents of an allocated block. If the block contained garbage values before allocation, it contains garbage after allocation, too.

The realloc Function

Sometimes you may need to adjust the size of an allocated memory block. The **realloc** (reallocate) function can expand or shrink an existing memory block. The function takes two arguments:

- The address of an existing allocated block
- The size (in bytes) you want to give the block

The realloc library function expands or shrinks an existing allocated block.

If enough memory is available to accommodate the resized block, **realloc** allocates sufficient memory and copies as much of the existing block as the new block will hold. If the new block is smaller than the original, data is truncated.

For instance, if you had allocated a 30-element **int** array with the statement

```
ptr = (int *) calloc( (size_t)30, (size_t)sizeof( int ) );
```

the following statement would expand the block to contain 20 extra elements, for a total of 50:

```
ptr = (int *)realloc( ptr, (size_t)sizeof( int ) * 50 );
```

The address you pass to **realloc** can be the address returned from a previous call to any memory-allocation function: **malloc**, **calloc**, or **realloc** itself.

Like **malloc**, both **calloc** and **realloc** return a null address if they fail. Remember to check the return value whenever you call a memory-allocating function.

Keeping Out of Trouble

Here are a few rules to help you avoid trouble when allocating memory dynamically:

- Always check the return value when allocating memory.
- Be careful not to index past the boundaries of an allocated memory block.
- Free allocated memory as soon as you have finished using it.
- Make sure that the address you pass to the **free** function is valid.
- Don't use a pointer to an allocated block after freeing the block.

Most of these points were mentioned earlier, but the second deserves a little elaboration. As you may recall from earlier chapters, the C language doesn't check array subscripts or pointer references for validity. It's important to remember this rule when using a pointer to access an allocated block.

For instance, suppose that you allocate a 30-element integer array with the statement

```
ptr = (int *) malloc( (size_t)sizeof( int ) * 30 );
```

and then execute either of these statements:

```
ptr[32] = 80;
*(ptr+32) = 80;
```

Since the array has only 30 elements, both of the latter statements overwrite memory outside the allocated memory block. The statements store the value 80 in the address four bytes (two **int** elements) above the highest element in the array.

While uncontrolled pointer operations always carry the potential for disaster, they can create especially tricky program bugs if you write just beyond an allocated memory block.

Near the beginning of each allocated block is a tiny "link" containing information about the block. The memory-allocating functions use these links to keep track of allocated memory, and the more blocks you have allocated, the more important it is to keep all the links intact. If a bad pointer reference overwrites a link, it can cause problems in an entirely unexpected part of your program.

CHAPTER 13

Graphics

This chapter explains how to call graphics functions that set points, draw lines, change colors, and draw shapes such as rectangles and circles. The first section lists the three steps to using high-resolution graphics, defines important graphics terms, and works through an example program step by step, showing how to use the basic functions. The next sections explain coordinate systems and show how to display graphics inside viewports and windows.

Graphics Mode

There are three steps to displaying graphics in QuickC:

1. Use the **_getvideoconfig** function to determine which video adapter is installed. (See the section "Checking the Current Video Mode.")
2. Use the **_setvideomode** function to set the desired graphics mode for the installed video adapter. (See the section "Setting the Video Mode.")
3. Draw the graphics on the screen. (See the section "Writing a Graphics Program.")

There are several definitions you need to know before you can create graphics programs. The following list explains the most useful terms:

- The "*x* axis" determines the horizontal position on the screen. The "origin" (point 0, 0) is in the upper left corner. The maximum number of horizontal "pixels" (picture elements) varies from 320 to 640 to 720, depending on the graphics card installed and the graphics mode in effect.
- The "*y* axis" is the vertical position. The origin is the upper left corner. The number of vertical pixels ranges from 200 to 480.

- Each graphics mode offers a "palette" from which you may choose the colors to be displayed. You may have access to 2, 4, 8, 16, or 256 "color indexes," depending on the graphics card in the computer and the graphics mode in effect.
- The CGA (Color Graphics Adapter) modes offer four fixed palettes containing predefined colors that may not be changed. In EGA (Enhanced Graphics Adapter), MCGA (Multicolor Graphics Array), and VGA (Video Graphics Array) graphics modes, you may change any of the color indexes by providing a color value that describes the mix of colors you wish to use.
- A color index is always a short integer. A color value is always a long integer. When you're calling graphics functions that require color-related parameters, you should be aware of the difference between color indexes and color values.

Checking the Current Video Mode

Before or after entering graphics mode, you may inquire about the current video configuration. This requires a special structure type called **videoconfig**, which is defined in the GRAPH.H header file. You pass the address of the structure to the function **_getvideoconfig**, which returns the current video configuration information.

All graphics programs should include the graphics header file and declare a structure of type **videoconfig**. The structure contains the following elements:

```
short numxpixels;     /*number of pixels on x axis*/
short numypixels;     /*number of pixels on y axis*/
short numtextcols;    /*number of text columns available*/
short numtextrows;    /*number of text rows available*/
short numcolors;      /*number of color indexes*/
short bitsperpixel;   /*number of bits per pixel*/
short numvideopages;  /*number of available video pages*/
short mode;           /*current video mode*/
short adapter;        /*active display adapter*/
short monitor;        /*active display monitor*/
short memory;         /*adapter video memory in K bytes*/
```

These variables within the **videoconfig** structure are initialized when you call **_getvideoconfig**.

Setting the Video Mode

Before you can start drawing pictures on the screen, your program must tell the graphics adapter to switch from video text mode to graphics mode. To do this, call **_setvideomode**, passing it a single integer that tells it which mode to display. The following constants are defined in the GRAPH.H file. The dimensions are listed in pixels for graphics mode and in columns for video text modes.

Constant	Video Mode	Mode Type/Hardware
_DEFAULTMODE	Restores to original mode	Both/All
_ERESCOLOR	640 × 350, 4 or 16 color	Graphics/EGA
_ERESNOCOLOR	640 × 350, BW	Graphics/EGA
_HERCMONO	720 × 348, BW for HGC	Graphics/HGC
_HRES16COLOR	640 × 200, 16 color	Graphics/EGA
_HRESBW	640 × 200, BW	Graphics/CGA
_MRES4COLOR	320 × 200, 4 color	Graphics/CGA
_MRES16COLOR	320 × 200, 16 color	Graphics/EGA
_MRES256COLOR	320 × 200, 256 color	Graphics/VGA/ MCGA
_MRESNOCOLOR	320 × 200, 4 gray	Graphics/CGA
_ORESCOLOR	640 × 400, 1 of 16 colors	Graphics/ Olivetti®
_TEXTBW40	40-column text, 16 gray	Text/CGA
_TEXTBW80	80-column text, 16 gray	Text/CGA
_TEXTC40	40-column text, 16/8 color	Text/CGA
_TEXTC80	80-column text, 16/8 color	Text/CGA
_TEXTMONO	80-column text, BW	Text/MDA
_VRES2COLOR	640 × 480, BW	Graphics/VGA/ MCGA
_VRES16COLOR	640 × 480, 16 color	Graphics/VGA

If **_setvideomode** returns a 0, it means the hardware does not support the selected mode. You may continue to select alternate video modes until a nonzero value is returned. If the hardware configuration doesn't support any of the selected video modes, take the appropriate exit action.

Writing a Graphics Program

The SINE.C program below graphs a sine curve. The program illustrates how to call many of the important graphics functions. The **main** function calls five other functions, which are defined later in this chapter. To view the complete program, use online help.

WARNING *When you installed QuickC on your system, you may have chosen not to include the graphics library. If this is the case, the programs in this chapter won't compile unless you explicitly link the graphics library. See the Microsoft QuickC Tool Kit for information about linking libraries.*

```
/* SINE.C: Basic graphics commands. */

#include <stdio.h>
#include <stdlib.h>
#include <graph.h>
#include <math.h>
#include <conio.h>
#define PI 3.14159

void graphics_mode( void );
void draw_lines( void );
void sine_wave( void );
void draw_shapes( void );
void end_program( void );
int newx( int );
int newy( int );

struct videoconfig myscreen;
int maxx, maxy;
unsigned char diagmask[8] =
{ 0x93, 0xC9, 0x64, 0xB2, 0x59, 0x2C, 0x96, 0x4B };
unsigned char linemask[8] =
{ 0xFF, 0x00, 0x7F, 0xFE, 0x00, 0x00, 0x00, 0xCC };


main()
{
   graphics_mode();
   draw_lines();
   sine_wave();
   draw_shapes();
   end_program();
}
/*
Definitions of functions go here
*/
```

The SINE.C program's output is shown in Figure 13.1.

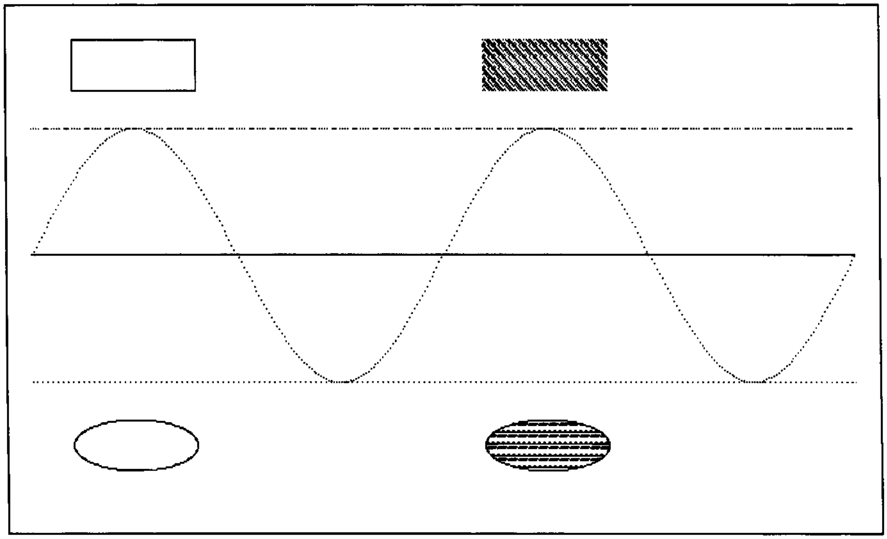


Figure 13.1 SINE.C Program

Turning on Graphics Mode

Before you can display graphics, you must put the graphics adapter into a graphics mode. The **_setvideomode** function performs this task. Before calling **_setvideomode**, you must decide which graphics modes are acceptable for your purposes. The first function in SINE.C is named `graphics_mode`. It selects the highest possible resolution available, based on the graphics card currently in use.

Four header files are included in the SINE.C program:

```
#include <stdio.h>

#include <stdlib.h>

#include <graph.h>

#include <math.h>
```

Although the MATH.H file is not required for graphics programs, we include it in the SINE.C program because it contains floating-point math functions such as **sin**.

Later in the program we'll need to get information about the screen size, so the **videoconfig** structure called `myscreen` is declared:

```
struct videoconfig myscreen;
```

The functions called by **main** aren't in the standard library; they're defined within SINE.C.

The first function is `graphics_mode`, which turns on graphics capabilities:

```
void graphics_mode( void )
{
   _getvideoconfig( &myscreen );
   switch( myscreen.adapter )
   {
      case _CGA:
         _setvideomode( _HRESBW );
         break;
      case _OCGA:
         _setvideomode( _ORESCOLOR );
         break;
      case _EGA:
      case _OEGA:
         if( myscreen.monitor == _MONO )
            _setvideomode( _ERESNOCOLOR );
         else
            _setvideomode( _ERESCOLOR );
         break;
      case _VGA:
      case _OVGA:
      case _MCGA:
         _setvideomode( _VRES2COLOR );
         break;
      case _HGC:
         _setvideomode( _HERCMONO );
         break;
      default:
         printf( "This program requires a CGA, EGA, VGA, or
Hercules card\n" );
         exit( 0 );
   }
   _getvideoconfig( &myscreen );
   maxx = myscreen.numxpixels - 1;
   maxy = myscreen.numypixels - 1;
}
```

NOTE *If you use a Hercules® adapter, you must run the MSHERC.COM program before attempting to display any graphics. Always run MSHERC.COM before running QuickC (do not run it from QuickC's DOS shell).*

The function begins by calling **_getvideoconfig**, passing the address of the **videoconfig** structure. Within the structure a member called **adapter** tells us the type of adapter currently in use. With that knowledge, and a switch statement, we can enter the appropriate graphics mode.

But how much screen do we have to work with? The screen might be 720 × 348, 640 × 480, 640 × 400, 640 × 350, or 640 × 200. Whenever you call **_setvideomode**, you can ask for information about the currently displayed screen with **_getvideoconfig**. Just pass it the address of the **videoconfig** structure that was declared earlier:

```
_getvideoconfig( &myscreen );
maxx = myscreen.numxpixels - 1;
maxy = myscreen.numypixels - 1;
```

Let's say your computer has an EGA card, which means that at this point, **_ERESNOCOLOR** is in effect. The horizontal screen size is 640 pixels and vertical screen size is 350. The two assignments above assign these values to `maxx` and `maxy`, less 1. The horizontal resolution might be 640, but the pixels are numbered 0–639. Thus, the `maxx` variable—the highest available pixel number—must be 1 less than the total number of pixels:

```
myscreen.numxpixels - 1
```

Two short functions perform conversions from an imaginary 1000 × 1000 screen to whatever graphics mode is in effect. From this point forward, the program will assume it has 1000 pixels in each direction, passing the values to `newx` and `newy` for conversion to actual coordinates:

```
int newx( int xcoord )
{
   int nx;
   float tempx;
   tempx = ((float)maxx)/ 1000.0;
   tempx = ((float)xcoord) * tempx + 0.5;
   return( (int)tempx );
}

int newy( int ycoord )
{
   int ny;
   float tempy;
   tempy = ((float)maxy)/ 1000.0;
   tempy = ((float)ycoord) * tempy + 0.5;
   return( (int)tempy );
}
```

Drawing Rectangles and Lines

The next function called in SINE.C is `draw_lines`. As the name implies, the `draw_lines` function draws several lines on the screen: a rectangle around the outer edges of the screen and three horizontal lines that cut the screen into quarters.

```
void draw_lines( void )
{
   _rectangle( _GBORDER, 0, 0, maxx, maxy );
   /* _setcliprgn( 20, 20, maxx - 20, maxy - 20 ); */
   _setvieworg( 0, newy( 500 ) );

   _moveto( 0, 0 );
   _lineto( newx( 1000 ), 0 );
   _setlinestyle( 0xAA3C );
   _moveto( 0, newy( -250) );
   _lineto( newx( 1000 ), newy( -250 ) );
   _setlinestyle( 0x8888 );
   _moveto( 0, newy( 250 ) );
   _lineto( newx( 1000 ), newy( 250 ) );
}
```

The call to the **_rectangle** function has five arguments. The first argument is the fill flag, which may be either **_GBORDER** or **_GFILLINTERIOR**. Choose **_GBORDER** if you want a rectangle of four lines (a border only, in the current line style). Or you can choose **_GFILLINTERIOR** if you want a solid rectangle (filled in with the current color and fill pattern). We will discuss how to choose the color and fill pattern later in this chapter.

The second and third arguments are the *x* and *y* coordinates of one corner of the rectangle. The fourth and fifth arguments are the coordinates of the opposite corner. Since the coordinates for the two corners are `( 0, 0 )` and `( maxx, maxy )`, the call to **_rectangle** frames the screen.

```
_rectangle( _GBORDER, 0, 0, maxx, maxy );
```

Drawing lines is a two-step process. Move to one location on the screen and draw the line to another location, using the **_moveto** and **_lineto** functions:

```
_setlinestyle( 0xAA3C );
_moveto( 0, newy(-250) );
_lineto( newx(1000), newy(-250) );
```

Use the **_setlinestyle** function to change from a solid line to a dashed line by passing it one integer value. In the example above, the number 0xAA3C causes the line to become the graphics equivalent of binary 1010 1010 0011 1100.

The **_moveto** function positions an imaginary pixel cursor at a spot on the screen. Nothing visible appears on the screen. The **_lineto** function draws a line. The negative value –250 might seem to be an impossible screen coordinate. It would be, but the program has changed the viewport organization of the screen with the **_setvieworg** function. The top half of the screen now contains negative *y* coordinates, and the bottom half contains positive *y* coordinates. Viewports are explained in more detail later in this chapter.

Setting a Pixel

The next step in the SINE.C program is to draw the sine curve. This requires the `sine_wave` function which is shown below. This function calculates positions for two sine waves and plots them on the screen:

```
void sine_wave( void )
{
 int locx, locy;
   double i, rad;

 for( i = 0; i < 1000; i += 3 )
   {
      rad = -sin( (PI * (float) i) / 250.0 );
      locx = newx( (int) i );
      locy = newy( (int) (rad * 250.0) );
      _setpixel( locx, locy );
   }
}
```

The only graphics function called is **_setpixel**, which takes two parameters, an *x* and a *y* coordinate. The function turns on the pixel at that location.

Drawing Shapes

After the sine curve is drawn, the SINE.C program calls the `draw_shapes` function to draw two rectangles and two ellipses on the screen. The fill mask alternates between **_GBORDER** and **_GFILLINTERIOR**:

```
void draw_shapes( void )
{
   _setlinestyle( 0xFFFF );
   _setfillmask( diagmask );
   _rectangle( _GBORDER, newx( 50 ), newy( -325 ), newx( 200 ), newy( -425 ) );
   _rectangle(_GFILLINTERIOR,newx(550),newy(-325),newx(700),newy(-425));

   _setfillmask( linemask );
   _ellipse( _GBORDER, newx( 50 ), newy( 325 ), newx( 200 ), newy( 425 ) );
   _ellipse( _GFILLINTERIOR,newx( 550 ),newy( 325 ),newx( 700 ),newy( 425 ) );
}
```

Note that **_setlinestyle** resets the line pattern to solid. If you omit this function (or comment it out), the first rectangle would be drawn with dashes instead of a solid line.

The **_ellipse** function draws an ellipse on the screen. Its parameters resemble the parameters for **_rectangle**. Both functions require a fill flag and two corners of a "bounding rectangle." When the ellipse is drawn, four points touch the edges of the bounding rectangle.

The **_GFILLINTERIOR** flag fills the shape with the current fill pattern. To select a pattern, you must first use the **_setfillmask** function, passing the address of an eight-byte array of unsigned characters. Earlier in the program `diagmask` was defined as the shape shown in Table 13.1 below.

Table 13.1 Fill Patterns

Bit Pattern	Value in diagmask
● ○ ○ ● ○ ○ ● ●	`diagmask[0]` = 0x93
● ● ○ ○ ● ○ ○ ●	`diagmask[1]` = 0xC9
○ ● ● ○ ○ ● ○ ○	`diagmask[2]` = 0x64
● ○ ● ● ○ ○ ● ○	`diagmask[3]` = 0xB2
○ ● ○ ● ● ○ ○ ●	`diagmask[4]` = 0x59
○ ○ ● ○ ● ● ○ ○	`diagmask[5]` = 0x2C
● ○ ○ ● ○ ● ● ○	`diagmask[6]` = 0x96
○ ● ○ ○ ● ○ ● ●	`diagmask[7]` = 0x4B

Exiting Graphics Mode

The final function to be called by the SINE.C program is `end_program`, which waits for a key press and then sets the screen back to normal:

```
void end_program( void )
{
   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Using Color Graphics Modes

In this example, the program COLOR.C sets a mode with as many color choices as possible for the available hardware:

```
/* COLOR.C: Sets a medium resolution mode
   with maximum color choices. */

#include <stdio.h>
#include <stdlib.h>
#include <graph.h>
#include <conio.h>
struct videoconfig vc;

main()
{
   if( _setvideomode( _MRES256COLOR ) );
   else if( _setvideomode( _MRES16COLOR ) );
   else if( _setvideomode( _MRES4COLOR ) );
   else
   {
      printf( "Error: No color graphics capability\n" );
      exit( 0 );
   }

   _getvideoconfig( &vc );

   printf( "%d available colors\n", vc.numcolors );
   printf( "%d horizontal pixels\n", vc.numxpixels );
   printf( "%d vertical pixels\n", vc.numypixels );

   getch();
   _clearscreen( _GCLEARSCREEN );
   _setvideomode( _DEFAULTMODE );
}
```

Although color graphics are an improvement over black and white, if you use color you must make a compromise. When you request the maximum number of colors, you sacrifice some resolution—a 320 × 200 screen instead of a higher resolution. Thus, the COLORS.C program always creates a screen 320 pixels wide and 200 pixels high. Note also the use of the function **_clearscreen**, which clears the screen in any video mode (text or graphics).

To view every possible graphics mode, you can run the program GRAPHIC.C shown below. Explanations of the various color graphics modes—CGA, EGA, and VGA—follow.

```
/* GRAPHIC.C: Display every graphics mode. */
#include <stdio.h>
#include <graph.h>
#include <conio.h>

struct videoconfig screen;
int modes[12] =
{
   _MRES4COLOR, _MRESNOCOLOR, _HRESBW, _HERCMONO,
   _MRES16COLOR, _HRES16COLOR, _ERESNOCOLOR, _ERESCOLOR,
   _VRES2COLOR, _VRES16COLOR, _MRES256COLOR, _ORESCOLOR
};

void print_menu( void );
void show_mode( char );

main()
{
   char key;
   print_menu();
   while( (key = getch()) != 'x' )
      show_mode( key );
}

void print_menu( void )
{
   _setvideomode( _DEFAULTMODE );
   _clearscreen( _GCLEARSCREEN );
   printf( "Please choose a graphics mode\nType 'x' to exit.\n\n" );
   printf( "0 _MRES4COLOR\n1 _MRESNOCOLOR\n2 _HRESBW\n" );
   printf( "3 _HERCMONO\n4 _MRES16COLOR\n5 _HRES16COLOR\n" );
   printf( "6 _ERESNOCOLOR\n7 _ERESCOLOR\n" );
   printf( "8 _VRES2COLOR\n9 _VRES16COLOR\na _MRES256COLOR\n" );
   printf( "b _ORESCOLOR\n" );
}

void show_mode( char which )
{
   int nc, i;
   int height, width;
   int mode = which;

   if( mode < '0' || mode > '9' )
      if( mode == 'a' )
    mode = '9' + 1;
      else if( mode == 'b' )
    mode = '9' + 2;
```

```
      else
     return;

   if( _setvideomode( modes[mode - 'Ø'] ) )
   {
      _getvideoconfig( &screen );
      nc = screen.numcolors;
      width = screen.numxpixels/nc;
      height = screen.numypixels/2;
      for( i = Ø; i < nc; i++ )
      {
    _setcolor( i );
    _rectangle( _GFILLINTERIOR, i * width, Ø, (i + 1) * width, height );
      }
   }
   else
   {
      printf( " \nVideo mode %c is not available.\n", which );
      printf( "Please press a key.\n" );
   }
   getch();
   _setvideomode( _DEFAULTMODE );
   print_menu();
}
```

CGA Color Graphics Modes

The CGA color graphics modes **_MRES4COLOR** and **_MRESNOCOLOR** display four colors selected from one of several predefined palettes of colors. They display these foreground colors against a background color which can be any one of the 16 available colors. With the CGA hardware, the palette of foreground colors is predefined and cannot be changed. Each palette number is an integer as shown in Table 13.2.

Table 13.2 Available CGA Colors

		Color Index	
Palette Number	**1**	**2**	**3**
0	Green	Red	Brown
1	Cyan	Magenta	Light gray
2	Light green	Light red	Yellow
3	Light cyan	Light magenta	White

The **_MRESNOCOLOR** graphics mode produces palettes containing various shades of gray on black-and-white monitors. The **_MRESNOCOLOR** mode displays colors when used with a color display. However, only two palettes

are available with a color display. You can use the **_selectpalette** function to select one of these predefined palettes. Table 13.3 shows the correspondence between the color indexes and the palettes.

Table 13.3 CGA Colors: _MRESNOCOLOR Mode

	Color Index		
Palette Number	**1**	**2**	**3**
0	Blue	Red	Light gray
1	Light blue	Light red	White

You may use the **_selectpalette** function only with the **_MRES4COLOR** and **_MRESNOCOLOR** graphics modes. To change palettes in other graphics modes, use the **_remappalette** or **_remapallpalette** functions.

The following program sets the video mode to **_MRES4COLOR** and then cycles through background colors and palette combinations. It works on computers equipped with CGA, EGA, MCGA, or VGA cards. A color monitor is required.

```
/* CGA.C: Demonstrate CGA colors. */

#include <stdio.h>
#include <graph.h>
#include <conio.h>

long bkcolor[8] =
{
   _BLACK, _BLUE, _GREEN, _CYAN,
   _RED, _MAGENTA, _BROWN, _WHITE
};

char *bkcolor_name[] =
{
   "_BLACK", "_BLUE", "_GREEN", "_CYAN",
   "_RED", "_MAGENTA", "_BROWN", "_WHITE"
};
```

```
main()
{
   int i, j, k;
   _setvideomode( _MRES4COLOR );
   for( i=0; i<= 3; i++ )
   {
      _selectpalette( i );
      for( k=0; k <= 7; k++ )
      {
         _setbkcolor( bkcolor[k] );
         for( j=0; j<=3; j++ )
         {
            _settextposition( 1, 1 );
            printf( "background color: %8s\n", bkcolor_name[k] );
            printf( "palette: %d\ncolor: %d\n", i, j );
            _setcolor( j );
            _rectangle( _GFILLINTERIOR, 160, 100, 320, 200 );
             getch();
         }
      }
   }
   _setvideomode( _DEFAULTMODE );
}
```

EGA, MCGA, and VGA Palettes

At the beginning of this chapter, we mentioned the difference between color indexes and color values. An analogy might make things clearer. Imagine a painter who owns 64 tubes of paint and a painter's palette that has room for only 16 globs of paint at any one time. A painting created under these constraints could contain only 16 colors (selected from a total of 64). One of the EGA graphics modes (**_ERESCOLOR**) is similar: 16 color indexes chosen from a total of 64 color values. (Color indexes are sometimes called "color attributes," or "pixel values." Color values are sometimes called "actual colors.")

VGA Color Mixing VGA offers the widest variety of color values: 262,144 (256K). Depending on the graphics mode, the VGA palette size may be 2, 16, or 256. When you select a color value, you specify a level of intensity ranging from 0–63 for each of the red, green, and blue color values. The long integer that defines a color value consists of four bytes (32 bits):

```
MSB                              LSB
zzzzzzzz zzBBBBBB zzGGGGGG zzRRRRRR
```

The most-significant byte must contain all zeros. The two high bits in the remaining three bytes must also be 0. To mix a light red (pink), turn red all the way up, and mix in some green and blue:

```
00000000 00100000 00100000 00111111
```

To represent this value in hexadecimal, use the number `0x0020203FL` (the `L` marks it as a long value). You could also use the following macro:

```
#define RGB ( r, g, b ) (0x3F3F3FL & ((long)(b) << 16 | (g) << 8 |
(r)))
```

To create pure yellow (100% red plus 100% green) and assign it to a variable `y1`, use this line:

```
y1 = RGB( 63, 63, 0 );
```

For white, turn all the colors on: `RGB( 63, 63, 63 )`. For black, set all colors to 0: `RGB( 0, 0, 0 )`.

EGA Color Mixing Mixing colors in EGA modes is similar to the mixing described above, but there are fewer intensities for the red, green, and blue components. In the modes that offer 64 colors, the R, G, and B values cover 2 bits and can range from 0 to 3. The long integer that defines an RGB color looks like this:

```
MSB                                 LSB
zzzzzzzz zzBB???? zzGG???? zzRR????
```

The bits marked `z` must be zeros and the bits marked with question marks can be any value. To form a pure red color value, you would use the constant `0x00000030L`. For cyan (blue plus green), use `0x00303000L`. The RGB macro defined above is easily modified for EGA monitors:

```
#define EGARGB( r, g, b ) (0x3F3F3FL & ((long)(b) << 20 | (g) << 12
| (r << 4)))
```

In this macro, you would pass values in the range 0–3 instead of 0–63.

EGA Color Graphics Modes

The _**MRES16COLOR**, _**HRES16COLOR**, or _**ERESCOLOR** video modes display the best color graphics with an EGA adapter. The CGA modes will also display on the EGA but with the lower CGA resolution and decreased color options.

The _**remappalette** function assigns a new color value to a color index. For example, when you first enter an EGA graphics mode, color index 1 equals the color value blue. To reassign the pure red color value to color index 1, you could use this line:

```
_remappalette( 1, 0x000030L );
```

Or, use the symbolic constant _**RED**, which is defined in the `GRAPH.H` file:

```
_remappalette( 1, _RED );
```

After this function call, any object currently drawn in color index 1 will instantly switch from blue to red.

For EGA graphics, the first value is an integer in the range 0–15 and the second value is a **long int** defined as a mixture of red, green, and blue (you may also use the symbolic constants such as _**RED**).

The _**remapallpalette** function changes all of the color indexes simultaneously. You pass it an array of color values. The first color value in the list becomes the new color associated with the color index 0.

The number in a function call to set the color (such as _**setcolor**) is an index into the palette of available colors. In the default text palette, an index of 1 refers to blue but the palette could be remapped to change index 1 to any other available color. As a result, the color produced by that pixel value also changes. The number of color indexes depends on the number of colors supported by the current video mode.

The _**remappalette** and _**remapallpalette** functions work in all modes but only with the EGA, MCGA, or VGA hardware. The _**remappalette** and _**remapallpalette** functions fail and return a value of –1 when you attempt to remap a palette without the EGA, MCGA, or VGA hardware.

The following program draws a rectangle with a red interior. In the default EGA palette, color index 4 is red. This color index is changed to **_BLUE** in this program.

```
/* EGA.C: EGA palettes. */

#include <stdio.h>
#include <conio.h>
#include <graph.h>

main()
{
   _setvideomode( _ERESCOLOR );
   _setcolor( 4 );
   _rectangle( _GFILLINTERIOR, 50, 50, 200, 200 );

   _settextposition( 1, 1 );
   printf( "Normal palette\n" );
   printf( "Press a key" );
   getch();

   _remappalette( 4, _BLUE );

   _settextposition( 1, 1 );
   printf( "Remapped palette\n" );
   printf( "Press a key" );
   getch();

   _remappalette( 4, _RED );

   _settextposition( 1, 1 );
   printf( "Restored palette\n" );
   printf( "Press a key to clear the screen" );
   getch();

   _clearscreen( _GCLEARSCREEN );
   _setvideomode( _DEFAULTMODE );
}
```

VGA Color Graphics Modes

The VGA card adds graphics modes **_VRES2COLOR**, **_VRES16COLOR**, and **_MRES256COLOR** to your repertoire. EGA and CGA modes can also be used with the VGA hardware, but with either lower resolution or fewer color choices.

The VGA color graphics modes operate with a range of 262,144 (256K) color values. The **_VRES2COLOR** graphics mode displays two colors, the **_VRES16COLOR** graphics mode displays 16, and the **_MRES256COLOR** graphics mode displays 256 colors from the available VGA colors.

Changing the Palette The **_remappalette** function changes a color index to a specified color value. The function below remaps the color index 1 to the color value given by the symbolic constant **_RED** (which represents red). After this statement is executed, whatever was displayed as blue will now appear as red:

```
_remappalette( 1, _RED );  /* reassign color index 1
                              to VGA red */
```

Use the **_remapallpalette** function to remap all of the available color indexes simultaneously. The function's argument references an array of color values that reflects the remapping. The first color number in the list becomes the new color associated with color index 0.

Symbolic constants for the default color numbers are supplied so that the remapping of VGA colors is compatible with EGA practice. The names of these constants are self-explanatory. For example, the color numbers for black, red, and light yellow are represented by the symbolic constants **_BLACK**, **_RED**, and **_LIGHTYELLOW**.

All of the VGA display modes operate with any VGA video monitor. Colors are displayed as shades of gray when the monochrome analog display is connected.

If you have a VGA card, the HORIZON.C program illustrates what can be done with the range of 256 colors:

```
/* HORIZON.C: VGA graphics with cycling of 256 colors. */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graph.h>

#define RED 0x0000003FL
#define GRN 0x00003F00L
#define BLU 0x003F0000L
#define WHT 0x003F3F3FL
#define STEP 21

struct videoconfig screen;
long int rainbow[512];

main()
{
   int i;
   long int col, gray;

   if( _setvideomode( _MRES256COLOR ) == 0 )
   {
      printf( "This program requires a VGA card.\n" );
      exit( 0 );
   }
   for( col = 0; col < 64; col++ )
   {
      gray = col | (col << 8) | (col << 16);
      rainbow[col] = rainbow[col + 256] = BLU & gray;
      rainbow[col + 64] = rainbow[col + 64 + 256] = BLU | gray;
      rainbow[col + 128] = rainbow[col + 128 + 256] = RED | (WHT & ~gray);
      rainbow[col + 192] = rainbow[col + 192 + 256] = RED & ~gray;
   }
   _setvieworg( 160, 85 );

   for( i = 0; i < 255; i++ )
   {
      _setcolor( 255 - i );
      _moveto( i, i - 255 );
      _lineto( -i, 255 - i );
      _moveto( -i, i - 255 );
      _lineto( i, 255 - i );
      _ellipse( _GBORDER, -i, -i / 2, i, i / 2 );
   }
   for( i = 0; !kbhit(); i += STEP, i %= 256 )
      _remapallpalette( &(rainbow[i]) );

   _setvideomode( _DEFAULTMODE );
}
```

Using the Color Video Text Modes

Two color video text modes, **_TEXTC40** and **_TEXTC80**, can be used with the CGA, EGA, and VGA displays. These modes display steady or blinking text in any of 16 foreground colors with any one of 8 background colors.

Basics of Text Color Selection

In a video text mode, each displayed character requires two bytes of video memory. The first byte contains the ASCII code representing the character and the second byte contains the display attribute. In the CGA color video text modes, the attribute byte determines the color and whether it will blink. Sixteen colors are available: the CGA pixel values, and the default EGA and VGA pixel values. Since the EGA and VGA palette can be remapped, these values can be made to correspond to any set of 16 colors with the appropriate palette mapping.

Using Text Colors

Use the **_gettextcolor** and **_getbkcolor** functions to find the current text foreground and background colors.

Values in the range 0–15 are interpreted as normal color. Values in the range 16–31 are the same colors as those in the range 0–15 but with blinking text.

Use the **_settextcolor** and **_setbkcolor** functions to set foreground and background colors in video text mode. These functions use a single argument that specifies the pixel value to be used for text displayed with the **_outtext** function. The color indexes for color video text modes are defined in Table 13.4.

Table 13.4 Text Colors

Number	Color	Number	Color
0	Black	8	Dark gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Light brown
7	White	15	Light white

Displaying Color Text

The **_settextposition** function moves the cursor to a row and column for displaying color text. The **_outtext** function displays the text.

Example: Viewing Text Colors

The following program displays a chart showing all possible combinations of text and background colors:

```
/* COLTEXT.C: Display text in color. */

#include <stdio.h>
#include <conio.h>
#include <graph.h>

char buffer [80];

main()
{
   int blink,fgd;
   long bgd;

   _clearscreen( _GCLEARSCREEN );
   printf( "Text color attributes:\n" );

   for( blink=0; blink<=16; blink+=16 )
   {
      for( bgd=0; bgd<8; bgd++ )
      {
         _setbkcolor( bgd );
         _settextposition( bgd + ((blink / 16) * 9) + 3, 1 );
         _settextcolor( 7 );
         sprintf( buffer, "Bgd: %d Fgd:", bgd );
         _outtext( buffer );

         for( fgd=0; fgd<16; fgd++ )
         {
            _settextcolor( fgd+blink );
            sprintf( buffer, " %2d ", fgd+blink );
            _outtext( buffer );
         }
      }
   }
   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Text Coordinates

Before you can write a program to print a word *over there* on the screen, you need a system that describes to the compiler where *there* really is. QuickC divides the text screen into rows and columns. See Figure 13.2.

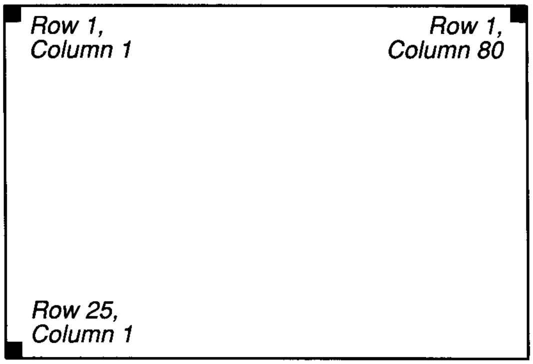


Figure 13.2 Text Screen Coordinates

Two important conventions to keep in mind about video text mode are:

1. Numbering starts at 1, not 0. An 80-column screen contains columns 1–80.
2. The row is always listed before the column.

If the screen is in a video text mode that displays 25 rows and 80 columns (as in Figure 13.2), the rows are numbered 1–25 and the columns are numbered 1–80. In functions such as **_settextposition**, which is called in the next example program, the parameters you pass are row and column (in that order).

Graphics Coordinates

A similar (but slightly different) system is used for locating pixels on a graphics screen. There are three ways of describing the location of pixels on the screen:

1. The physical screen coordinates
2. The viewport coordinates
3. The window coordinates

Each method is explained in the following sections.

The Physical Screen

Suppose you write a program that calls **_setvideomode** and puts the screen into the VGA graphics mode **_VRES16COLOR**. This gives you a screen containing 640 horizontal pixels and 480 vertical pixels. The individual pixels are named by their location relative to the *x* axis and *y* axis, as shown in Figure 13.3.

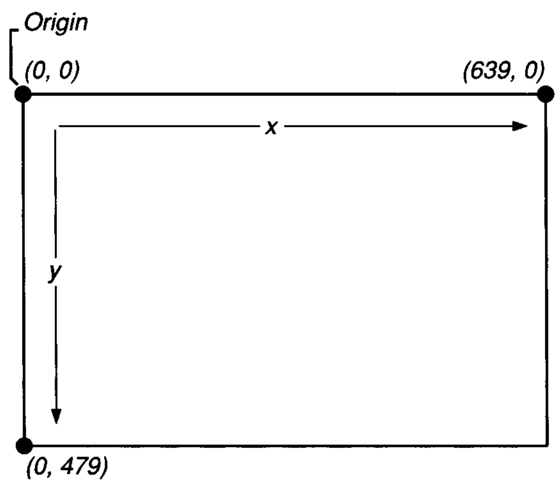


Figure 13.3 Physical Screen Coordinates

Two important differences between text coordinates and pixel coordinates are:

1. Numbering starts at 0, not 1. If there are 640 pixels, they're numbered 0–639.
2. The *x* coordinate (equivalent to a text column) is listed before the *y* coordinate.

The upper left corner is called the "origin." The *x* and *y* coordinates for the origin are always (0, 0). If you use variables to refer to pixel locations, declare them as integers.

Changing the Origin with _setvieworg

The **_setvieworg** function changes the current location of the viewport's origin. When you first enter graphics mode, the "viewport" is equivalent to the physical

screen. You pass two integers, which represent the *x* and *y* coordinates of a physical screen location. For example, the following line would move the origin to the physical screen location (50, 100):

```
_setvieworg( 50, 100 );
```

The effect on the screen is illustrated in Figure 13.4.

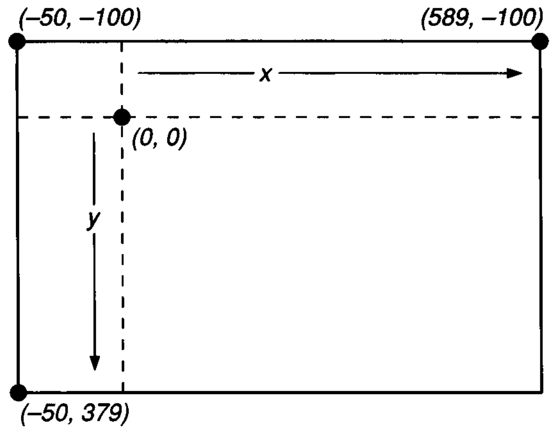


Figure 13.4 Coordinates Changed by _setvieworg

The number of pixels hasn't changed, but the names given to the points have changed. The *x* axis now ranges from –50 to +589 instead of 0 to 639. The *y* axis now covers the values –100 to +379. (If you own an adapter other than the VGA, the numbers are different but the effect is the same.)

All standard graphics functions are affected by the new origin, including **_arc**, **_ellipse**, **_lineto**, **_moveto**, **_pie**, and **_rectangle**.

For example, if you call the **_rectangle** function after relocating the viewport origin, and pass it the values (0, 0) and (40, 40), the rectangle would be drawn 50 pixels from the left edge of the screen and 100 pixels from the top. It would not appear in the upper left corner.

The values passed to **_setvieworg** are always physical screen locations. Suppose you called the same function twice:

```
_setvieworg( 50, 100 );
_setvieworg( 50, 100 );
```

The viewport origin would not move to (100, 200). It would remain at the physical screen location (50, 100).

Defining a Clipping Region with _setcliprgn

The **_setcliprgn** function creates an invisible rectangular area on the screen called a "clipping region." Attempts to draw inside the clipping region are successful, while attempts to draw outside the region are not.

When you first enter a graphics mode, the default clipping region occupies the entire screen. QuickC ignores any attempts to draw outside the screen.

Changing the clipping region requires one call to **_setcliprgn**. Suppose you've entered the CGA graphics mode **_MRES4COLOR**, which has a screen resolution of 320 × 200. If you draw a diagonal line from (0, 0) to (319, 199), from the top left to the bottom right corner, the screen looks like Figure 13.5.

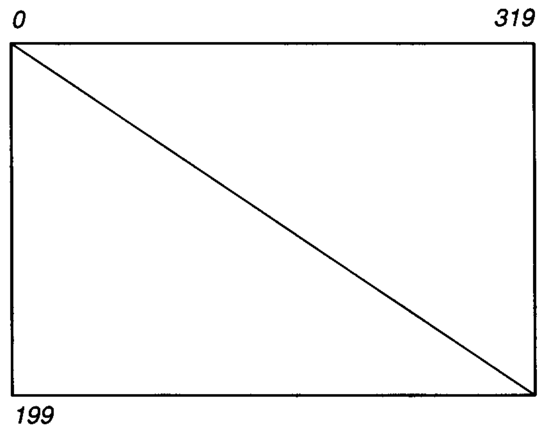


Figure 13.5 Line Drawn on a Full Screen

You could create a clipping region with this line:

```
_setcliprgn( 10, 10, 309, 189 )
```

With the clipping region in effect, the same **_lineto** command would put the line shown in Figure 13.6 on the screen.

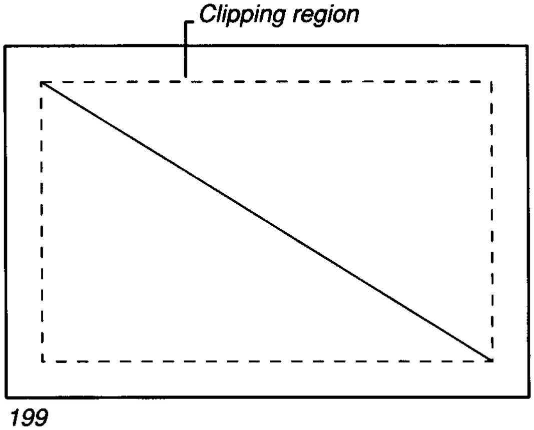


Figure 13.6 Line Drawn within a Clipping Region

The broken lines don't actually print on the screen. They indicate the outer bounds of the clipping region.

Viewport Coordinates

The **_setviewport** function establishes a new viewport within the boundaries of the physical screen. A standard viewport has two distinguishing features:

1. The origin of a viewport is in the upper left corner.
2. The clipping region matches the outer boundaries of the viewport.

The **_setviewport** function does the same thing as calling the **_setvieworg** and the **_setcliprgn** functions.

Real Coordinates in a Window

Functions that refer to coordinates on the physical screen and within the viewport require integer values. In real-life graphing applications, you might wish to use floating-point values—stock prices, the price of wheat, average rainfall, and so on. The **_setwindow** function allows you to scale the screen to almost any size. In addition, the window-related functions take double-precision, floating-point values instead of integers.

For example, say you want to graph 12 months of average temperatures that range from –40 to +100. You could add the following line to your program:

```
_setwindow( TRUE, 1.0, -40.0, 12.0, 100.0 );
```

The first argument is the invert flag, which puts the lowest *y* value in the bottom left corner. The minimum and maximum Cartesian coordinates follow (the decimal point marks them as floating-point values). The new organization of the screen is shown in Figure 13.7.

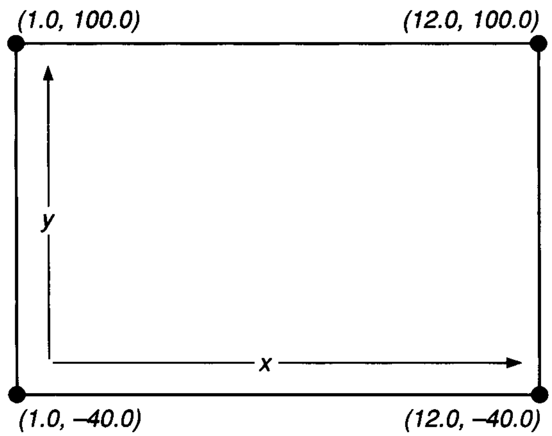


Figure 13.7 Window Coordinates

Note that January and December are plotted on the left and right edges of the screen. In an application like this, it might be better to number the *x* axis from 0.0 to 13.0, to provide some extra space.

If you next plot a point with **_setpixel_w** or draw a line with **_lineto_w**, the values are automatically scaled to the established window.

Follow these four steps to use real-coordinate graphics:

1. Enter a graphics mode with **_setvideomode**.
2. Use **_setviewport** to create a viewport area. (This step is optional if you plan to use the entire screen.)
3. Create a real-coordinate window with **_setwindow**, passing an **int** invert flag and four **double** *x* and *y* coordinates for the minimum and maximum values.
4. Draw graphics shapes with **_rectangle_w** and other functions. Do not confuse **_rectangle** (the viewport function) with **_rectangle_w** (the window function for drawing rectangles). All window functions end with an underscore and a letter **w** or an underscore and **wxy**.

Real-coordinate graphics can give you a lot of flexibility. For example, you can fit either axis into a small range (such as 151.25 to 151.45) or into a large range (–50,000 to +80,000), depending on the type of data you're graphing. In addition, by changing the window coordinates, you can create the effects of zooming in or panning across a figure.

Example Program

The program below illustrates some ways to use the real-coordinate windowing functions.

```
/* REALG.C: Real-coordinate graphics. */

#include <stdio.h>
#include <conio.h>
#include <graph.h>

#define TRUE 1
#define FALSE 0

int four_colors( void );
void three_graphs( void );
void grid_shape( void );

int halfx, halfy;
struct videoconfig screen;
double bananas[] =
   {
     -0.3, -0.2, -0.224, -0.1, -0.5, +0.21, +2.9,
     +0.3, +0.2, 0.0, -0.885, -1.1, -0.3, -0.2,
     +.001, +.005, +0.14, 0.0, -0.9, -0.13, +0.3
   };

main()
{
   if( four_colors() )
      three_graphs();
   else
      printf( "This program requires a CGA, EGA,\
               or VGA graphics card.\n" );
}
/*
. Additional functions defined below
.
.
*/
```

The **main** function is very short. It calls the `four_colors` function (defined below), which attempts to enter a graphics mode where at least four colors are available. If it succeeds, the `three_graphs` function is called, which uses the

numbers in the `bananas` array to draw three graphs. The REALG.C screen output is shown in Figure 13.8.

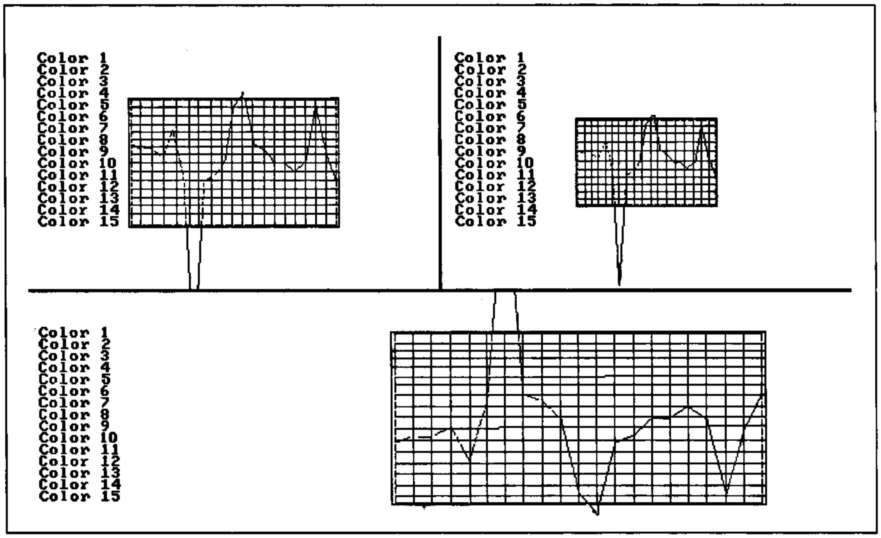


Figure 13.8 REALG.C Program

It's worth noting that the `grid_shape` function (defined below) that draws the graphs is using the same numbers in each case. However, the program uses three different real-coordinate windows. The two windows in the top half are the same size in physical coordinates, but they have different window sizes. In all three cases, the grid is 2 units wide. In the upper left corner, the window is 4 units wide; in the upper right, the window is 6 units wide, which makes the graph appear smaller.

In two of the three graphs, one of the lines goes off the edge, outside the clipping region. The lines do not intrude into the other windows, since defining a window creates a clipping region.

Finally, note that the graph on the bottom of the screen seems to be upside down with respect to the two graphs above it.

Checking the Adapter

The first step in any graphics program is to enter a graphics mode. The `four_colors` function performs this step:

```
/* four_colors function from REALG.C. */

int four_colors( void )
{
   _getvideoconfig( &screen );
   switch( screen.adapter )
   {
      case _CGA:
      case _OCGA:
         _setvideomode( _MRES4COLOR );
         break;
      case _EGA:
      case _OEGA:
         _setvideomode( _ERESCOLOR );
         break;
      case _VGA:
      case _OVGA:
         _setvideomode( _VRES16COLOR );
         break;
      default:
         return( FALSE );
   }
   _getvideoconfig( &screen );
   return( TRUE );
}
```

The **_getvideoconfig** function places some information into the **videoconfig** structure called `screen`. Then we use the member `screen.adapter` in a **switch** statement construct to turn on the matching graphics mode. The symbolic constants **_CGA** and the rest are defined in the GRAPH.H file. The modes that begin with the letter **O** are Olivetti modes.

If the computer is equipped with a color card, **_getvideoconfig** returns a **TRUE**. If it is not, it returns a **FALSE**, which causes **main** to skip the `three_graphs` function.

Three Windows, Three Graphs

If the `four_colors` function works properly, **main** calls the function below, which prints the three graphs.

```
/* three_graphs function from REALG.C. */

void three_graphs( void )
{
   int xwidth, yheight, cols, rows;
   struct _wxycoord upleft, botright;

   _clearscreen( _GCLEARSCREEN );
   xwidth = screen.numxpixels;
   yheight = screen.numypixels;
   halfx = xwidth/2;
   halfy = yheight/2;
   cols = screen.numtextcols;
   rows = screen.numtextrows;

   /* first window */
   _setviewport( 0, 0, halfx-1, halfy-1 );
   _settextwindow( 1, 1, rows/2, cols/2 );
   _setwindow( FALSE, -2.0, -2.0, 2.0, 2.0 );
   grid_shape();
   _rectangle( _GBORDER, 0, 0, halfx-1, halfy-1 );

   /* second window */
   _setviewport( halfx, 0, xwidth-1, halfy-1 );
   _settextwindow( 1, cols/2+1, rows/2, cols );
   _setwindow( FALSE, -3.0, -3.0, 3.0, 3.0 );
   grid_shape();
   _rectangle_w( _GBORDER, -3.0, -3.0, 3.0, 3.0 );

   /* third window */
   _setviewport( 0, halfy, xwidth-1, yheight-1 );
   _settextwindow( rows/2+1, 1, rows, cols );
   _setwindow( TRUE, -3.0, -1.5, 1.5, 1.5 );
   grid_shape();
   upleft.wx = -3.0;
   upleft.wy = -1.5;
   botright.wx = 1.5;
   botright.wy = 1.5;
   _rectangle_wxy( _GBORDER, &upleft, &botright );

   getch();
   _setvideomode( _DEFAULTMODE );
}
```

Clearing the Screen Although entering a graphics mode automatically clears the screen, it doesn't hurt to be sure, so `three_graphs` calls the **_clearscreen** function:

```
_clearscreen( _GCLEARSCREEN );
```

The **_GCLEARSCREEN** constant causes the entire physical screen to clear. Other options include **_GVIEWPORT** and **_GWINDOW**, which clear the current viewport and the current text window, respectively.

The First Window After assigning values to some variables, the `three_graphs` function creates the first window:

```
_setviewport( 0, 0, halfx - 1, halfy - 1 );
_settextwindow( 1, 1, rows / 2, cols / 2 );
_setwindow( FALSE, -2.0, -2.0, 2.0, 2.0 );
```

First a viewport is defined to cover the upper left quarter of the screen. Next, a text window is defined within the boundaries of that border. (Note the numbering starts at 1 and the row location precedes the column.) Finally, a window is defined. The **FALSE** constant forces the *y* axis to increase from top to bottom. The corners of the window are (–2.0, –2.0) in the upper left and (2.0, 2.0) in the bottom right corner.

Next, the function `grid_shape` is called, and a border is added to the window:

```
grid_shape();
_rectangle( _GBORDER, 0, 0, halfx-1, halfy-1 );
```

Note that this is the standard **_rectangle** function, which takes coordinates relative to the viewport (not window coordinates).

Two More Windows The two other windows are similar to the first. All three call `grid_shape` (defined below), which draws a grid from location (–1.0, –1.0) to (+1.0, +1.0). The grid appears in different sizes because the coordinates in the windows vary. The second window ranges from (–3.0, –3.0) to (+3.0, +3.0), so the width of the grid is one-third the width of the second window, while it is one-half the width of the first.

Note also that the third window contains a **TRUE** as the first argument. This causes the *y* axis to increase from bottom to top, instead of top to bottom. As a result, this graph appears to be upside down in relation to the other two.

After calling grid_shape, the program frames each window with one of the following functions:

```
_rectangle( _GBORDER, 0, 0, halfx -1, halfy -1 );

_rectangle_w( _GBORDER, -3.0, -3.0, 3.0, 3.0 );

_rectangle_wxy( _GBORDER, &upleft, &botright );
```

All three functions contain a fill flag as the first argument. The **_rectangle** function takes integer arguments that refer to the viewport screen coordinates. The function **_rectangle_w** takes four double-precision, floating-point values referring to window coordinates: upper left *x*, upper left *y*, lower right *x*, and lower right *y*. The function **_rectangle_wxy** takes two arguments: the addresses of two structures of type **_wxycoord**, which contains two **double** types named **wx** and **wy**. The structure is defined in GRAPH.H. The values are assigned just before **_rectangle_wxy** is called.

Text, Colors, and Lines

The grid_shape function is shown below:

```
/* grid_shape from the REALG.C program. */

void grid_shape( void )
{
   int i, numc, x1, y1, x2, y2;
   double x, y;
   char txt[80];

numc = screen.numcolors;
   for( i = 1; i < numc; i++ )
   {
      _settextposition( i, 2 );
      _settextcolor( i );
      sprintf( txt, "Color %d", i );
      _outtext( txt );
   }
   _setcolor( 1 );
   _rectangle_w( _GBORDER, -1.0, -1.0, 1.0, 1.0 );
   _rectangle_w( _GBORDER, -1.02, -1.02, 1.02, 1.02 );
```

```
    for( x = -0.9, i = 0; x < 0.9; x += 0.1 )
    {
       _setcolor( 2 );
       _moveto_w( x, -1.0 );
       _lineto_w( x, 1.0 );
       _moveto_w( -1.0, x );
       _lineto_w( 1.0, x );

       _setcolor( 3 );
       _moveto_w( x - 0.1, bananas[i++] );
       _lineto_w( x, bananas[i] );
    }
    _moveto_w( 0.9, bananas[i++] );
    _lineto_w( 1.0, bananas[i] );
}
```

First, the number of available color indexes is assigned to the `numc` variable and a **for** loop displays all of the available colors:

```
numc = screen.numcolors;
for( i = 1; i < numc; i++ )
{
   _settextposition( i, 2 );
   _settextcolor( i );
   sprintf( txt, "Color %d", i );
   _outtext( txt );
}
```

The names of the functions are self-explanatory. The advantage of using **_outtext** in graphics mode is that, unlike **printf**, you can control the text color.

The function names that end with **_w** work the same as their viewport equivalents, except you pass double-precision, floating-point values instead of integers. For example, you pass integers to **_lineto** but floating-point values to **_lineto_w.**

If you're interested in further explorations of graphics, Chapters 14 and 15 introduce Presentation Graphics and fonts, both of which offer even more graphics options.

CHAPTER 14

Presentation Graphics

Presentation Graphics is the name given to a library of chart-generating functions included with the QuickC package. With these functions your QuickC programs can display data as a variety of graphs such as pie charts, bar and column charts, line graphs, and scatter diagrams. Whole columns of unintelligible numbers can be reduced to a single expressive picture with Presentation Graphics.

This chapter shows you how to use the Presentation Graphics library in your QuickC programs. The first section is an introduction to Presentation Graphics. It explains terminology and describes some of the library's many capabilities. The middle sections of this chapter list the steps involved in writing a charting program and illustrate them with short examples.

The concluding portions of the chapter delve more deeply into Presentation Graphics. Here you'll learn about the Presentation Graphics default data structures and how to manipulate them. The final section presents a short reference list of all the functions that comprise the Presentation Graphics library.

To use Presentation Graphics you need a graphics adapter and a monitor capable of bit-mapped display—the same equipment mentioned in Chapter 13, "Graphics." Support is provided for CGA, EGA, VGA, MCGA, Hercules monochrome graphics, and Olivetti Color Board.

Terminology

Certain terms and phrases pertaining to Presentation Graphics and its functions are used throughout this chapter. The following description of Presentation Graphics terminology will help you better understand this chapter.

Data Series

Groups or series of data can be graphed on the same chart.

Data that are related by a common idea or purpose constitutes a "series." For example, the prices of a futures commodity over the course of a year form a single series of data. The commodity's volume and open interest form two more series for the same period of time. Presentation Graphics allows you to plot multiple series on the same graph. In theory only your system's memory capacity restricts the number of data series that can appear on a graph. However, there are practical considerations.

Characteristics such as color and pattern help distinguish one series from another. You can more readily differentiate series on a color monitor than you can on a monochrome monitor. The number of series that can comfortably appear on the same chart depends on the chart type and the number of available colors. Only experimentation can tell you what is best for your system.

Categories

Categories are non-numeric data. A set of categories forms a frame of reference for the comparisons of numeric data. For example, the months of the year are categories against which numeric data such as rainfall can be plotted.

Regional sales provide another example. A chart can show comparisons of a company's sales in different parts of the country. Each region forms a category. The sales within each region are numeric data that have meaning only within the context of a particular category.

Values

Values are numeric data. Sales, stock prices, air temperatures, populations—all are series of values that can be plotted against categories or against other values.

Presentation Graphics allows you to overlay different series of value data on a single graph. For example, average monthly temperatures or monthly sales of heating oil during different years—or a combination of temperatures and sales—can be plotted together on the same graph.

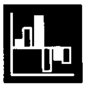
Pie Charts

"Pie charts" are used to represent data by showing the relationship of each part to the whole. A good example is a company's monthly sales figures. The sales to the company's various accounts can be represented as slices of the pie.

Presentation Graphics can display either a standard or an "exploded" pie chart. The exploded view shows the pie with one or more pieces separated for emphasis. Presentation Graphics optionally labels each slice of a pie chart with a percentage figure.

Bar and Column Charts

As the name implies, a "bar chart" shows data as horizontal bars. Bar charts show comparisons among items rather than absolute value.



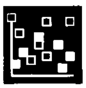
"Column charts" are vertical bar charts. Column charts are frequently used to show variations over a period of time, since they suggest time flow better than a bar chart

Line Graphs

"Line graphs" illustrate trends or changes in data. They show how a series of values varies against some category—for example, average temperatures throughout a particular year.

Traditionally, line graphs show a collection of data points connected by lines; hence the name. However, Presentation Graphics can also plot points that are not connected by lines.

Scatter Diagrams



A "scatter diagram" is the only type of graph available in Presentation Graphics that compares values with values. A scatter diagram simply plots points. One value may correspond to several other values.

Scatter diagrams illustrate the relationship between numeric values in different groups of data. They graphically show trends and correlations not easily detected from rows and columns of raw numbers. This explains why scatter diagrams are a favorite tool of statisticians and forecasters.

They are most useful with relatively large populations of data. Consider, for example, the relationship between personal income and family size. If you poll one thousand wage earners for their income and family size, you have a scatter diagram with one thousand points. If you combine your results so that you're left with one average income for each family size, you have a line graph.

Axes

All Presentation Graphics charts except pie charts are displayed with two perpendicular reference lines called "axes." The vertical or *y* axis runs from top to bottom of the chart and is placed against the left side of the screen. The horizontal or *x* axis runs from left to right across the bottom of the screen.

The chart type determines which axes are used for category and value data.

The *x* axis is the category axis for column and line charts and the value axis for bar charts. The *y* axis is the value axis for column and line charts and the category axis for bar charts.

Chart Windows

The "chart window" defines that part of the screen on which the chart is drawn. Normally the window fills the entire screen, but Presentation Graphics allows you to resize the window for smaller graphs. By redefining the chart window to different screen locations, you can view separate graphs together on the same screen.

Data Windows

While the chart window defines the entire graph including axes and labels, the "data window" defines only the actual plotting area. This is the portion of the graph to the right of the *y* axis and above the *x* axis. You cannot directly specify the size of the data window. Presentation-Graphics automatically determines its size based on the dimensions of the chart window.

Chart Styles

Each of the five types of Presentation Graphics charts can appear in two different chart styles, as described in Table 14.1.

Table 14.1 Presentation Graphics Chart Styles

Chart Type	Chart Style #1	Chart Style #2
Pie	With percentages	Without percentages
Bar	Side-by-side	Stacked
Column	Side-by-side	Stacked
Line	Points with lines	Points only
Scatter	Points with lines	Points only

Bar and column charts have only one style when displaying a single series of data. The styles "side-by-side" and "stacked" are applicable when more than one series appear on the same chart. The first style arranges the bars or columns for the different series side by side, showing relative heights or lengths. The stacked style, illustrated in Figure 14.1 for a column chart, emphasizes relative sizes between bars or columns.

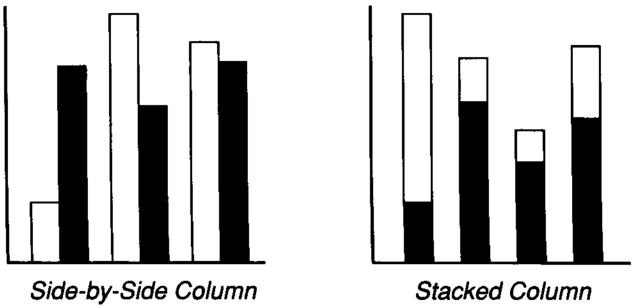


Figure 14.1 Side-by-Side and Stacked Styles for Typical Column Chart

Legends

Legends help identify individual data series.

When displaying more than one data series on a chart, Presentation Graphics uses different colors, line styles, or patterns to differentiate the series. Presentation Graphics also can display a "legend" that labels the different series of a chart. For a pie chart, the legend labels individual slices of the pie.

The format is similar to the legends found on printed graphs and maps. A sample of the color and pattern used to graph the series appears next to the series label. This identifies which set of data the labels belong to. The "Palettes" section later in this chapter explains how different data series are identified by color and pattern.

Presentation Graphics Program Structure

QuickC programs that use Presentation Graphics typically follow seven steps:

Step	Comments
Include required header files.	Along with other header files your program may need, you must include the files GRAPH.H and PGCHART.H.
Set video mode to graphics.	Refer to Chapter 13, "Graphics," for a discussion of video modes supported by QuickC. This chapter explains how to change modes within a QuickC program.
Initialize Presentation Graphics chart environment.	Presentation Graphics places charting parameters in a data structure. These parameters determine how a graph will appear on the screen. Collectively they make up the "chart environment," described in the section "Customizing Presentation Graphics." Presentation Graphics sets the environment parameters to default values. The amount of initialization that must be done by your program depends on how extensively it relies on defaults.
Assemble plot data.	Data can be collected in a variety of ways: by calculating it elsewhere in the program, reading it from files, or entering it from the keyboard. All plot data must be assembled in arrays because the Presentation Graphics functions locate them through pointers.
Call Presentation Graphics functions.	Display your chart.
Pause while chart is on the screen.	Your program should pause after a chart is displayed. This step allows sufficient time to read the chart. A common method is to wait for a keyboard entry before resuming.
Reset video mode.	When your program detects the signal to continue, it should normally reset the video to its original mode.

Once your program successfully compiles, you must link it to the library modules PGCHART.LIB and GRAPHICS.LIB. Use the Microsoft Overlay Linker

QLINK.EXE or the QCL command-line interface to link programs outside the QuickC environment. For descriptions of QLINK and QCL, see the *Microsoft QuickC Tool Kit*, Chapter 1, "Creating Executable Programs."

Five Example Chart Programs

You'll have a better idea of Presentation Graphics capabilities once you've seen what it can do. To that end some simple examples are presented in this section. The sample programs that follow use only five of the 22 Presentation Graphics functions: **_pg_initchart**, **_pg_defaultchart**, **_pg_chartpie**, **_pg_chart**, and **_pg_chartscatter**. Appendix B, "C Library Guide," and online help document these functions and their arguments. But the example code is straightforward, and you should be able to follow easily for now. Each program is commented so that you can recognize the seven steps given above.

A Sample Data Set

Suppose a grocer wants to graph the sales of orange juice over the course of a single year. Sales figures are on a monthly basis, so the grocer selects as category data the months of the year from January through December. The sales figures are shown below.

Month	Quantity (cases)
January	33
February	27
March	42
April	64
May	106
June	157
July	182
August	217
September	128
October	62
November	43
December	36

Example: Pie Chart

The following program uses Presentation Graphics to display a pie chart for the grocer's data. The chart, which is shown in Figure 14.2, remains on the screen until a key is pressed.

The Presentation Graphics functions return values that identify error conditions. A return value of 0 indicates that the function has completed its work without error. Refer to the header file PGCHART.H and online help for descriptions of the nonzero error codes.

```
/* PIE.C:  Create sample pie chart.  */

#include <conio.h>
#include <string.h>
#include <graph.h>
#include <pgchart.h>

#define MONTHS 12

typedef enum {FALSE, TRUE} boolean;

float far value[MONTHS] =
{
    33.0, 27.0, 42.0, 64.0,106.0,157.0,
   182.0,217.0,128.0, 62.0, 43.0, 36.0
};
char far *category[MONTHS] =
{
   "Jan", "Feb", "Mar", "Apr",
   "May", "Jun", "Jly", "Aug",
   "Sep", "Oct", "Nov", "Dec"
};
short far explode[MONTHS] = {0};

main()
{
   chartenv env;
   int mode = _VRES16COLOR;

   /* Set highest video mode available */
   while( !_setvideomode( mode ) )
      mode--;
   if( mode == _TEXTMONO )
      return( 0 );

   /* Initialize chart library and a default pie chart */
   _pg_initchart();
   _pg_defaultchart( &env, _PG_PIECHART, _PG_PERCENT );

   /* Add titles and some chart options */
   strcpy( env.maintitle.title, "Good Neighbor Grocery" );
   env.maintitle.titlecolor = 6;
   env.maintitle.justify = _PG_RIGHT;
   strcpy( env.subtitle.title, "Orange Juice Sales" );
   env.subtitle.titlecolor = 6;
   env.subtitle.justify = _PG_RIGHT;
   env.chartwindow.border = FALSE;
```

```
   /* Parameters for call to _pg_chartpie are:
    *
    *    env        - Environment variable
    *    category   - Category labels
    *    value      - Data to chart
    *    explode    - Separated pieces
    *    MONTHS     - Number of data values
    */
   if( _pg_chartpie( &env, category, value,
                     explode, MONTHS ) )
   {
      _setvideomode( _DEFAULTMODE );
      _outtext( "Error:  can't draw chart" );
   }
   else
   {
      getch();
      _setvideomode( _DEFAULTMODE );
   }
   return( Ø );
}
```

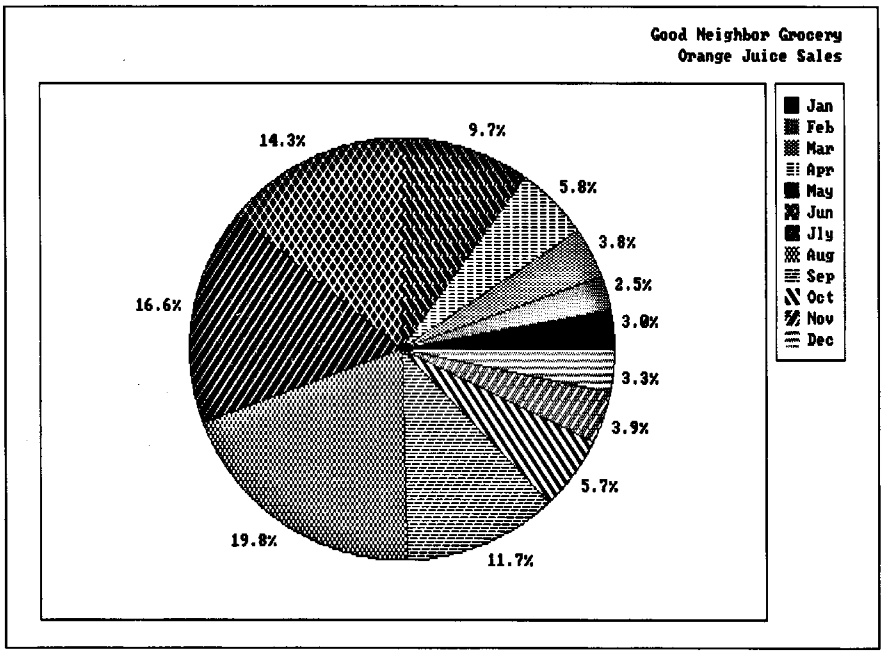


Figure 14.2 Example Pie Chart

Example: Bar Chart

The code for the PIE.C program needs only minor alterations to produce bar, column, and line charts for the same data:

- Replace the call to **_pg_chartpie** with **_pg_chart**. This function produces bar, column, and line charts depending on the value of the second argument for **_pg_defaultchart**.
- Give new arguments to **_pg_defaultchart** that specify chart type and style.
- Assign titles for the *x* axis and *y* axis in the structure `env`.
- Remove references to array `explode` (applicable only to pie charts).

The following example produces the bar chart shown in Figure 14.3.

```
/* BAR.C:  Create sample bar chart. */
#include <conio.h>
#include <string.h>
#include <graph.h>
#include <pgchart.h>
#define MONTHS 12
typedef enum {FALSE, TRUE} boolean;
float far value[MONTHS] =
{
    33.0, 27.0, 42.0, 64.0,106.0,157.0,
   182.0,217.0,128.0, 62.0, 43.0, 36.0
};
char far *category[MONTHS] =
{
   "Jan", "Feb", "Mar", "Apr",
   "May", "Jun", "Jly", "Aug",
   "Sep", "Oct", "Nov", "Dec"
};

main()
{
   chartenv env;
   int mode = _VRES16COLOR;
   /* Set highest video mode available */
   while( !_setvideomode( mode ) )
      mode--;
   if( mode == _TEXTMONO )
      return( 0 );

   /* Initialize chart library and a default bar chart */
   _pg_initchart();
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );

   /* Add titles and some chart options */
   strcpy( env.maintitle.title, "Good Neighbor Grocery" );
```

```
    env.maintitle.titlecolor = 6;
    env.maintitle.justify = _PG_RIGHT;
    strcpy( env.subtitle.title, "Orange Juice Sales" );
    env.subtitle.titlecolor = 6;
    env.subtitle.justify = _PG_RIGHT;
    strcpy( env.yaxis.axistitle.title, "Months" );
    strcpy( env.xaxis.axistitle.title, "Quantity (cases)" );
    env.chartwindow.border = FALSE;

    /* Parameters for call to _pg_chart are:
     *    env         - Environment variable
     *    category    - Category labels
     *    value       - Data to chart
     *    MONTHS      - Number of data values */
    if( _pg_chart( &env, category, value, MONTHS ) )
    {
       _setvideomode( _DEFAULTMODE );
       _outtext( "Error:  can't draw chart" );
    }
    else
    {
       getch();
       _setvideomode( _DEFAULTMODE );
    }
    return( 0 );
}
```

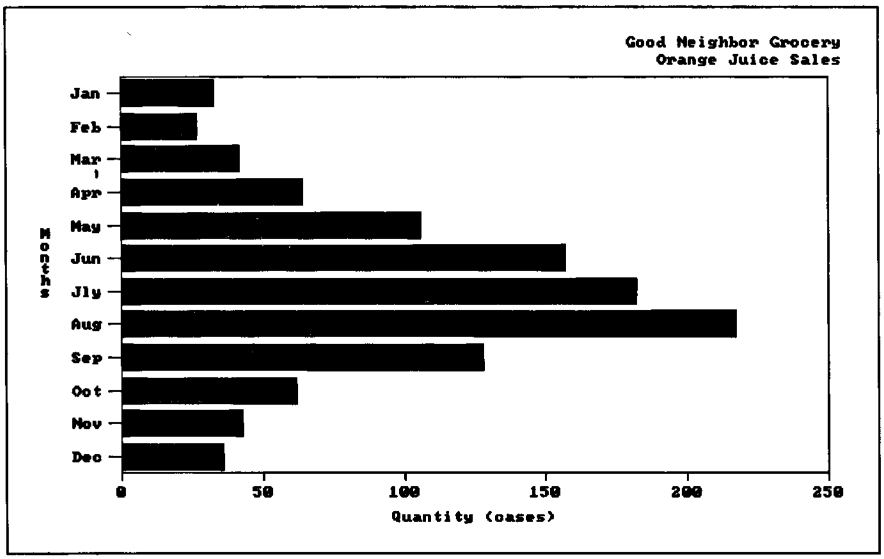


Figure 14.3 Example Bar Chart

Example: Column Chart

The grocer's bar chart becomes a column chart in two easy steps. Simply specify the new chart type when calling **_pg_defaultchart** and switch the axis titles. To produce a column chart for the data, replace the call to **_pg_defaultchart** with:

```
_pg_defaultchart( &env, _PG_COLUMNCHART, _PG_PLAINBARS );
```

and replace the last two calls to **strcpy** with:

```
strcpy( env.xaxis.axistitle.title, "Months" );
strcpy( env.yaxis.axistitle.title, "Quantity (cases)" );
```

Notice that now the *x* axis is labeled "Months" and the *y* axis is labeled "Quantity (cases)." Figure 14.4 shows the resulting column chart.

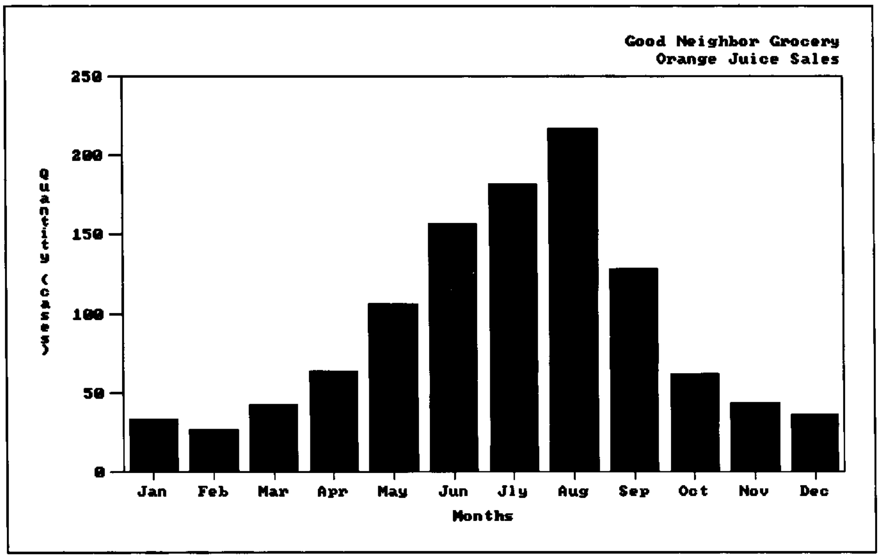


Figure 14.4 Example Column Chart

Example: Line Chart

Creating an equivalent line chart requires only one change. Use the same code as for the column chart and replace the call to **_pg_defaultchart** with:

```
_pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
```

Figure 14.5 shows the line chart for the grocer's data.

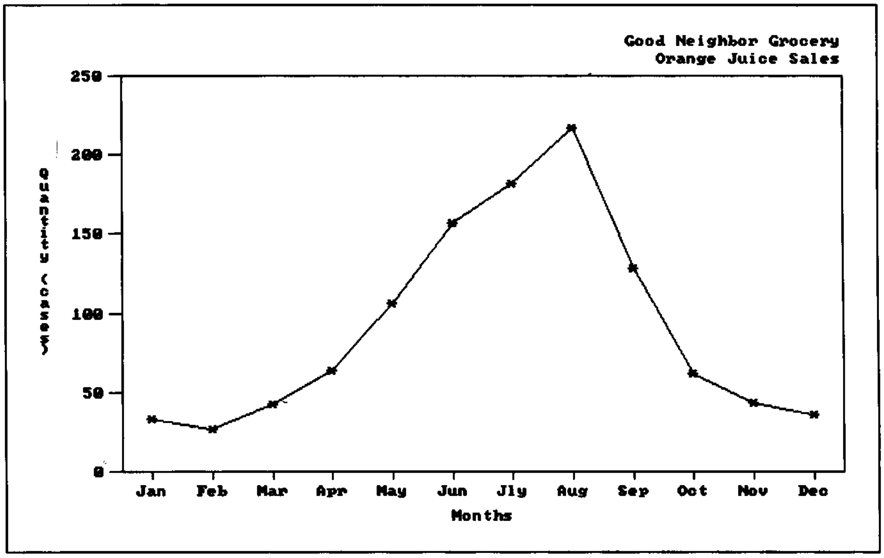


Figure 14.5 Example Line Chart

Example: Scatter Diagram

Now suppose that the store owner wants to compare the sales of orange juice to the sales of another product, say hot chocolate. Possible monthly sales are shown below.

Months	Orange Juice (cases)	Hot Chocolate (cases)
January	33	37
February	27	37
March	42	30
April	64	19
May	106	10
June	157	5
July	182	2
August	217	1
September	128	7
October	62	15
November	43	28
December	36	39

The program SCATTER.C displays a scatter diagram that illustrates the relationship between the sales of orange juice and hot chocolate throughout a 12-month period.

```
/* SCATTER.C:  Create sample scatter diagram. */

#include <conio.h>
#include <string.h>
#include <graph.h>
#include <pgchart.h>

#define MONTHS 12
typedef enum {FALSE, TRUE} boolean;

/* Orange juice sales */

float far xvalue[MONTHS] =
{
    33.0, 27.0, 42.0, 64.0,106.0,157.0,
   182.0,217.0,128.0, 62.0, 43.0, 36.0
};
/* Hot chocolate sales */

float far yvalue[MONTHS] =
{
   37.0, 37.0, 30.0, 19.0, 10.0,  5.0,
    2.0,  1.0,  7.0, 15.0, 28.0, 39.0
};

main()
{
   chartenv env;
   int mode = _VRES16COLOR;

   /* Set highest video mode available */

   while( !_setvideomode( mode ) )
      mode--;
   if( mode == _TEXTMONO )
      return( 0 );
```

```
    /* Initialize chart library and default
     * scatter diagram
     */
    _pg_initchart();
    _pg_defaultchart( &env, _PG_SCATTERCHART,
                      _PG_POINTONLY );

    /* Add titles and some chart options */

    strcpy( env.maintitle.title, "Good Neighbor Grocery" );
    env.maintitle.titlecolor = 6;
    env.maintitle.justify = _PG_RIGHT;
    strcpy( env.subtitle.title,
            "Orange Juice vs Hot Chocolate" );
    env.subtitle.titlecolor = 6;
    env.subtitle.justify = _PG_RIGHT;
    env.yaxis.grid = TRUE;
    strcpy( env.xaxis.axistitle.title,
            "Orange Juice Sales" );
    strcpy( env.yaxis.axistitle.title,
            "Hot Chocolate Sales" );
    env.chartwindow.border = FALSE;

    /* Parameters for call to _pg_chartscatter are:
     *    env        - Environment variable
     *    xvalue     - X-axis data
     *    yvalue     - Y-axis data
     *    MONTHS     - Number of data values
     */

    if( _pg_chartscatter( &env, xvalue,
                          yvalue, MONTHS ) )
    {
       _setvideomode( _DEFAULTMODE );
       _outtext( "Error:  can't draw chart" );
    }
    else
    {
       getch();
       _setvideomode( _DEFAULTMODE );
    }
    return( 0 );
}
```

Figure 14.6 shows the results of SCATTER.C. Notice that the scatter points form a slightly curved line, indicating a correlation exists between the sales of the two products. The store owner can conclude from the scatter diagram that the demand for orange juice is roughly inverse to the demand for hot chocolate.

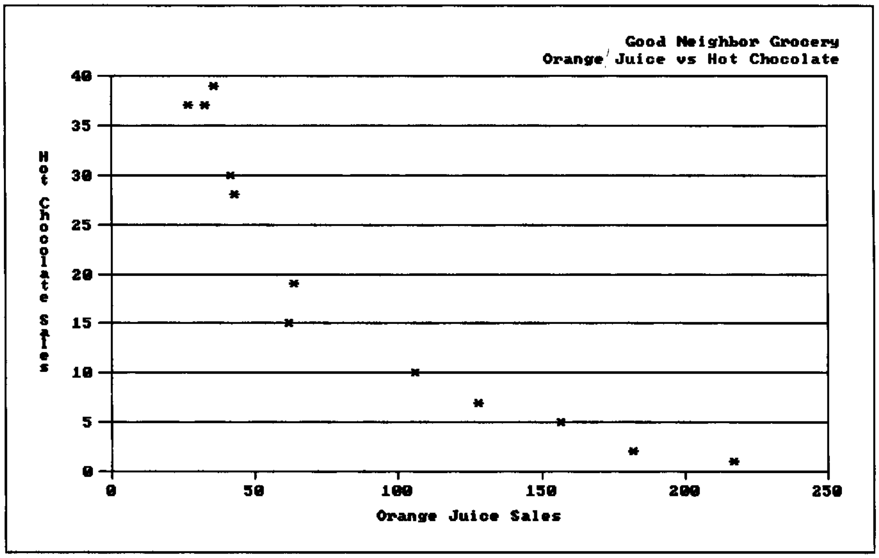


Figure 14.6 Example Scatter Diagram

Palettes

Presentation Graphics displays each data series in a way that makes it discernible from other series. It does this by defining a separate "palette" for every data series in a chart. Palettes consist of entries that determine color, line style, fill pattern, and plot character used to graph the series.

Presentation Graphics maintains its palettes as an array of structures. The header file PGCHART.H defines the palette structures as:

```
/* Typedef for pattern bitmap */
typedef unsigned char fillmap[8];

/* Typedef for palette entry definition */
typedef struct
{
   unsigned short color;
   unsigned short style;
   fillmap        fill;
   char           plotchar;
} paletteentry;

/* Typedef for palette definition */
typedef paletteentry palettetype[_PG_PALETTELEN];
```

It's important not to confuse the Presentation Graphics palettes with the adapter display palettes, which are register values kept by the video controller. The function **_selectpalette** described in Chapter 13, "Graphics," sets the display palette. It does not define the data series palettes used by Presentation Graphics.

Color Pool

Presentation Graphics organizes all chart colors into a "color pool." The color pool consists of pixel values valid for the current graphics mode. (Refer to Chapter 13, "Graphics," or the Glossary for a definition of pixel values.) Palette structures contain color codes that refer to the color pool. A palette's color code determines the color used to graph the data series associated with the palette. Colors of labels, titles, legends, and axes are also determined by the contents of the color pool.

The first element of the color pool is always 0, which is the pixel value for the screen background color. The second element is always the highest pixel value available for the graphics mode. The remaining elements are repeating sequences of available pixel values, beginning with 1.

As shown above, the first member of a palette data structure is:

```
unsigned short color;
```

This variable defines the color code for the data series associated with the palette. The color code is neither a display attribute nor a pixel value. It is an index number of the color pool.

An example should make this clearer. A graphics mode of **_MRES4COLOR** (320 × 200 graphics) provides four colors for display. Pixel values from 0 to 3 determine the possible pixel colors—say, black, green, red, and brown respectively. In this case the first 8 elements of the color pool would be the following:

Color Pool Index	Pixel Value	Color
0	0	Black
1	3	Brown
2	1	Green
3	2	Red
4	3	Brown
5	1	Green
6	2	Red
7	3	Brown

Notice that the sequence of available foreground colors repeats from the third element. The first data series in this case would be plotted in brown, the second series in green, the third series in red, the fourth series again in brown, and so forth.

Video adapters such as the EGA or the Hercules InColor™ Card allow 16 on-screen colors. This allows Presentation Graphics to graph more series without duplicating colors.

Style Pool

Presentation Graphics matches the color pool with a collection of different line styles called the "style pool." Entries in the style pool define the appearance of lines such as axes and grids. Lines can be solid, dotted, dashed, or of some combination.

The second member of a palette structure defines a style code as:

```
unsigned short style;
```

Each palette contains a style code that refers to an entry in the style pool in the same way that it contains a color code that refers to an entry in the color pool. The style code value in a palette is applicable only to line graphs and lined scatter diagrams. The style code determines the appearance of the lines drawn between points.

The palette's style code adds further variety to the lines of a multiseries graph. It is most useful when the number of lines in a chart exceeds the number of available colors. For example, a graph of nine different data series must repeat colors if only three foreground colors are available for display. However, the style code for each color repetition will be different, ensuring that none of the lines looks the same.

Pattern Pool

Presentation Graphics also maintains a pool of "fill patterns." Patterns determine the fill design for column, bar, and pie charts. The third member of a palette structure holds the palette's fill pattern. The pattern member is an array:

```
fillmap fill;
```

where `fillmap` is type-defined as:

```
typedef unsigned char fillmap[8];
```

Each fill pattern array holds an 8 × 8 bit map that defines the fill pattern for the data series associated with the palette. Table 14.2 shows how a fill pattern of diagonal stripes is created with the `fill` pattern array.

The bit map below corresponds to screen pixels. Each of the 8 layers of the map are binary numbers, where a solid circle signifies 1 and an open circle signifies 0. Thus the first layer of the map—that is, the first byte—represents the binary number 10011001, which is the decimal number 153.

Table 14.2 Fill Patterns

Bit Map	Value in fill
● ○ ○ ● ● ○ ○ ●	`fill[0]` = 153
● ● ○ ○ ● ● ○ ○	`fill[1]` = 204
○ ● ● ○ ○ ● ● ○	`fill[2]` = 102
○ ○ ● ● ○ ○ ● ●	`fill[3]` = 51
● ○ ○ ● ● ○ ○ ●	`fill[4]` = 153
● ● ○ ○ ● ● ○ ○	`fill[5]` = 204
○ ● ● ○ ○ ● ● ○	`fill[6]` = 102
○ ○ ● ● ○ ○ ● ●	`fill[7]` = 51

If you wish to create the above pattern for your chart's first data series, you must reset the `fill` array for the first palette structure. You can do this in five steps:

1. Declare a structure of type **palettetype** to hold the palette parameters.
2. Call **_pg_initchart** to initialize the palettes with default values.
3. Call the Presentation Graphics function **_pg_getpalette** to retrieve a copy of the current palette data.
4. Assign the values given in Table 14.2 to the array `fill` for the first palette.
5. Call the Presentation Graphics function **_pg_setpalette** to load the modified palette values.

The following lines of code demonstrate these five steps:

```
/* Declare a structure array for palette data. */

palettetype palette_struct;
.
.
.
/* Initialize chart library */

_pg_initchart();
.
.
.
/* Copy current palette data into palette_struct */

_pg_getpalette( palette_struct );

/* Reinitialize fill pattern for first palette using
    values in Table 14.2 */

palette_struct[1].fill[0] = 153;
palette_struct[1].fill[1] = 204;
palette_struct[1].fill[2] = 102;
palette_struct[1].fill[3] =  51;
palette_struct[1].fill[4] = 153;
palette_struct[1].fill[5] = 204;
palette_struct[1].fill[6] = 102;
palette_struct[1].fill[7] =  51;

/* Load new palette data */

_pg_setpalette( palette_struct );
```

Now when you display your bar or column chart the first series appears filled with the striped pattern shown in Table 14.2.

Pie charts are a bit different. The idea of multiple series does not really apply to them. Instead, palette structures correspond to individual slices. If the number of slices exceeds the constant **_PG_PALETTELEN**, palettes are recycled. Thus the first palette dictates not only the appearance of the first slice, but of slice number **_PG_PALETTELEN** as well. The second palette determines the appearance of both the second slice and of slice number **_PG_PALETTELEN** + 1, and so forth.

Character Pool

The last member of a palette structure is an index number in a pool of ASCII characters:

```
char plotchar;
```

The member `plotchar` represents plot points on line graphs and scatter diagrams. Each palette uses a different character to distinguish plot points between data series.

Customizing Presentation Graphics

Presentation Graphics is built for flexibility. You can use its system of default values to produce professional-looking charts with a minimum of programming effort. Or you can fine-tune the appearance of your charts by overriding default values and initializing variables explicitly in your program. The following section shows you how.

Chart Environment

The header file PGCHART.H defines a structure type **chartenv**. This structure type organizes the set of variables known as the "chart environment." The chart environment describes everything about a chart except the plots themselves. It's the blank page, in other words, ready for plotting data. The environment determines the appearance of text, axes, grid lines, and legends.

Calling the **_pg_defaultchart** function fills the chart environment with default values. Presentation Graphics allows you to reset any variable in the environment before displaying a chart. Except for adjusting the palette values, all initialization of data is done through a **chartenv** type structure.

The sample chart programs provided earlier illustrate how to adjust variables in the chart environment. These programs create a structure `env` of the type **chartenv**. The structure `env` contains the chart environment variables, initialized by the call to **_pg_defaultchart**. Environment variables such as the chart title are then given specific values, as in:

```
strcpy( env.maintitle.title, "Good Neighbor Grocery" );
```

Environment variables that determine colors and line styles deserve special mention. The chart environment holds several such variables, which can be recognized by their names. For example, the variable *titlecolor* specifies the color of title text. Similarly, the variable *gridstyle* specifies the line style used to draw the chart grid.

Colors and line styles in the chart environment are taken from palettes.

These variables are index numbers, but do not refer directly to the color pool or line pool. They correspond instead to palette numbers. If you set *titlecolor* to 2, Presentation Graphics uses the color code in the second palette to determine the title's color. Thus the title in this case would be the same color as the chart's second data series. If you change the color code in the palette, you'll also change the title's color.

A structure of type **chartenv** consists of four secondary structures. The file PGCHART.H type-defines the secondary structures as:

```
titletype
axistype
windowtype
legendtype
```

The remainder of this section describes the chart environment of Presentation Graphics. It first examines structures of the four secondary types that make up the chart environment structure. The section concludes with a description of the **chartenv** structure type. Each discussion begins with a brief explanation of the structure's purpose, followed by a listing of the structure type definition as it appears in the PGCHART.H file. All symbolic constants are defined in the file PGCHART.H.

titletype

Structures of type **titletype** determine text, color, and placement of titles appearing in the graph. The PGCHART.H file defines the structure type as:

```
typedef struct
{
  char     title[_PG_TITLELEN];  /* Title text */
  short    titlecolor;           /* Palette color
                                    for title text */
  short    justify;              /* _PG_LEFT, _PG_CENTER,
                                    _PG_RIGHT */
} titletype;
```

The following list describes **titletype** members:

Member Variable	Description
justify	An integer specifying how the title is justified within the chart window. The symbolic constants defined in the PGCHART.H file for this variable are **_PG_LEFT**, **_PG_CENTER**, and **_PG_RIGHT**.
titlecolor	An integer between 1 and **_PG_PALETTELEN** that specifies a title's color. The default value for *titlecolor* is 1.
title[**_PG_TITLELEN**]	A character array containing title text. For example, if `env` is a structure of type **chartenv**, then `env.maintitle.title` holds the character string used for the main title of the chart. Similarly, `env.xaxis.axistitle.title` contains the axis title. The number of characters in a title must be one less than **_PG_TITLELEN** to allow room for a null terminator.

axistype

Structures of type **axistype** contain variables for the axes such as color, scale, grid style, and tick marks. The PGCHART.H file defines the structure type as:

```
typedef struct
{
  short      grid;          /* TRUE=grid lines drawn;
                               FALSE=no lines */
  short      gridstyle;     /* Style bytes for grid */
  titletype  axistitle;     /* Title definition
                               for axis */
  short      axiscolor;     /* Color for axis */
  short      labeled;       /* TRUE=ticks marks and titles
                               drawn */
  short      rangetype;     /* _PG_LINEARAXIS,
                               _PG_LOGAXIS */
  float      logbase;       /* Base used if log axis */
  short      autoscale;     /* TRUE=next 7 values
                               calculated by system */
  float      scalemin;      /* Minimum value of scale */
  float      scalemax;      /* Maximum value of scale */
  float      scalefactor;   /* Scale factor for data on
                               this axis */
  titletype  scaletitle;    /* Title definition for
                               scaling factor */
  float      ticinterval;   /* Distance between tick marks
                               (world coord.) */
  short      ticformat;     /* _PG_EXPFORMAT or
                               _PG_DECFORMAT */
  short      ticdecimals;   /* Number of decimals for tick
                               labels (max=9) */
} axistype;
```

The following list describes **axistype** member variables:

Member Variable	Description
autoscale	A boolean variable. If *autoscale* is **TRUE**, Presentation Graphics automatically determines values for *scalefactor*, *scalemax*, *scalemin*, *scaletitle*, *ticdecimals*, *ticformat*, and *ticinterval* (see below). If *autoscale* equals **FALSE**, these seven variables must be specified in your program.
axiscolor	An integer between 1 and **_PG_PALETTELEN** that specifies the color used for the axis and parallel grid lines. (See description for *gridstyle* above.) Note that this member does not determine the color of the axis title. That selection is made through the structure *axistitle*.

axistitle
A **titletype** structure that defines the title of the associated axis. The title of the *y* axis displays vertically to the left of the *y* axis, and the title of the *x* axis displays horizontally below the *x* axis.

grid
A boolean true/false value that determines whether grid lines are drawn for the associated axis. Grid lines span the data window perpendicular to the axis.

gridstyle
An integer between 1 and **_PG_PALETTELEN** that specifies the grid's line style. Lines can be solid, dashed, dotted, or some combination. The default value for *gridstyle* is 1. Note that the color of the parallel axis determines the color of the grid lines. Thus the *x* axis grid is the same color as the *y* axis, and the *y* axis grid is the same color as the *x* axis.

labeled
A boolean value that determines whether tick marks and labels are drawn on the axis. Axis labels should not be confused with axis titles. Axis labels are numbers or descriptions such as "23.2" or "January" attached to each tick mark.

logbase
If *rangetype* is logarithmic, the *logbase* variable determines the log base used to scale the axis. Default value is 10.

rangetype
An integer that determines whether the scale of the axis is linear or logarithmic. The *rangetype* variable applies only to value data.

Specify a linear scale with the **_PG_LINEARAXIS** constant. A linear scale is best when the difference between axis minimum and maximum is relatively small. For example, a linear axis range 0–10 results in 10 tick marks evenly spaced along the axis.

Use **_PG_LOGAXIS** to specify a logarithmic *rangetype*. Logarithmic scales are useful when the range is very large or when the data varies exponentially. Line graphs of exponentially varying data can be made straight with a logarithmic *rangetype*.

scalefactor
All numeric data are scaled by dividing each value by *scalefactor*. For relatively small values, the variable *scalefactor* should be 1, which is the default. But data with large values should be scaled by an appropriate factor. For example, data in the range 2 million–20 million should be plotted with *scalemin* set to 2, *scalemax* set to 20, and *scalefactor* set to 1 million.

If *autoscale* is set to **TRUE**, Presentation Graphics automatically determines a suitable value for *scalefactor* based on the range of data to be plotted. Presentation Graphics selects only values that are a factor of 1 thousand—that is, values such as 1 thousand, 1 million, or 1 billion. It then labels the *scaletitle* appropriately (see below). If you desire some other value for scaling, you must set *autoscale* to **FALSE** and set *scalefactor* to the desired scaling value.

scalemax — Highest value represented by the axis.

scalemin — Lowest value represented by the axis.

scaletitle — A **titletype** structure defining a string of text that describes the value of *scalefactor*. If *autoscale* is **TRUE**, Presentation Graphics automatically writes a scale description to *scaletitle*. If *autoscale* equals **FALSE** and *scalefactor* is 1, *scaletitle.title* should be blank. Otherwise your program should copy an appropriate scale description to *scaletitle.title*, such as "(× 1000)," "(in millions of units)," "times 10 thousand dollars," etc.

For the *y* axis, the *scaletitle* text displays vertically between the axis title and the *y* axis. For the *x* axis, the scale title appears below the *x* axis title.

ticdecimals — Number of digits to display after the decimal point in tick labels. Maximum value is 9. Note that this variable applies only to axes with value data. It is ignored for the category axis.

ticformat — An integer that determines the format of the labels assigned to each tick mark. Set *ticformat* to **_PG_EXPFORMAT** for exponential format or set it to **_PG_DECFORMAT** for decimal. The default is **_PG_DECFORMAT**. Note that this variable applies only to axes with value data. It is ignored for the category axis.

ticinterval — Sets interval between tick marks on the axis. The tick interval is measured in the same units as the numeric data associated with the axis. For example, if 2 sequential tick marks correspond to the values 20 and 25, the tick interval between them is 5. Note that this variable applies only to axes with value data. It is ignored for the category axis.

windowtype

Structures of type **windowtype** contain sizes, locations, and color codes for the three windows produced by Presentation Graphics: the chart window, the data window, and the legend. Refer to the "Terminology" section at the beginning of this chapter for definitions of these terms. Windows are located on the screen relative to the screen's logical origin. By changing the logical origin, you can display charts that are partly or completely off the screen. The PGCHART.H file defines **windowtype** as:

```
typedef struct
{
  short  x1;           /* Left edge of window in
                          pixels */
  short  y1;           /* Top edge of window in
                          pixels */
  short  x2;           /* Right edge of window in
                          pixels */
  short  y2;           /* Bottom edge of window in
                          pixels */
  short  border;       /* TRUE for border, FALSE
                          otherwise */
  short  background;   /* Internal palette color for
                          window background */
  short  borderstyle;  /* Style bytes for window
                          border */
  short  bordercolor;  /* Internal palette color for
                          window border */
} windowtype;
```

The following list describes **windowtype** member variables:

Member Variable	Description
x1, *y1*, *x2*, *y2*	Window coordinates in pixels. The ordered pair (*x1*, *y1*) specifies the coordinate of the upper left corner of the window. The ordered pair (*x2*, *y2*) specifies the coordinate of the lower right corner. The reference point for the coordinates depends on the type of window. The chart window is located relative to the logical origin, usually the upper left corner of the screen. The data and legend windows are located relative to the upper left corner of the chart window. This allows you to change the position of the chart window without having to redefine coordinates for the other two windows.

background	An integer between 1 and **_PG_PALETTELEN** that specifies the window's background color. The default value for *background* is 1.
border	A boolean variable that determines whether a border frame is drawn around a window.
bordercolor	An integer between 1 and **_PG_PALETTELEN** that specifies the color of the window's border frame. The default value is 1.
borderstyle	An integer between 1 and **_PG_PALETTELEN** that specifies the line style of the window's border frame. The default value is 1.

legendtype

Structures of type **legendtype** contain size, location, and colors of the chart legend. The PGCHART.H file defines the structure type as:

```
typedef struct
{
  short      legend;        /* TRUE=draw legend;
                               FALSE=no legend */
  short      place;         /* _PG_RIGHT, _PG_BOTTOM,
                               _PG_OVERLAY */
  short      textcolor;     /* Palette color for text*/
  short      autosize;      /* TRUE=system calculates
                               legend size */
  windowtype legendwindow;  /* Window definition for
                               legend */
} legendtype;
```

The following list describes **legendtype** member variables:

Member Variable	Description
autosize	A boolean true/false variable that determines whether Presentation Graphics is to automatically calculate the size of the legend. If *autosize* equals **FALSE**, the legend window must be specified in the *legendwindow* structure (see below).
legend	A boolean true/false variable that determines whether a legend is to appear on the chart. The *legend* variable is ignored by functions that graph single-series charts.

legendwindow — A **windowtype** structure that defines coordinates, background color, and border frame for the legend. Coordinates given in *legendwindow* are ignored if *autosize* is **TRUE.**

place — An integer that specifies the location of the legend relative to the data window. Setting the variable *place* equal to the constant **_PG_RIGHT** positions the legend to the right of the data window. Setting *place* to **_PG_BOTTOM** positions the legend below the data window. Setting *place* to **_PG_OVERLAY** positions the legend within the data window.

These settings influence the size of the data window. If *place* is equal to **_PG_BOTTOM** or **_PG_RIGHT**, Presentation Graphics automatically sizes the data window to accommodate the legend. If *place* equals **_PG_OVERLAY** the data window is sized without regard to the legend.

textcolor — An integer between 1 and **_PG_PALETTELEN** that specifies the color of text within the legend window.

chartenv

A structure of type **chartenv** defines the chart environment. The following code shows that a **chartenv** type structure consists almost entirely of structures of the four types discussed above.

The PGCHART.H file defines the **chartenv** structure type as:

```
typedef struct
{
  short       charttype;     /* Chart type */
  short       chartstyle;    /* Chart style */
  windowtype  chartwindow;   /* Window definition for
                                overall chart */
  windowtype  datawindow;    /* Window definition for data
                                part of chart */
  titletype   maintitle;     /* Main chart title */
  titletype   subtitle;      /* Chart subtitle */
  axistype    xaxis;         /* Definition for x axis */
  axistype    yaxis;         /* Definition for y axis */
  legendtype  legend;        /* Definition for legend */
} chartenv;
```

Initialize the chart environment with the _pg_defaultchart function.

Note that all the data in a **chartenv** type structure is initialized by calling the **_pg_defaultchart** function. If your program does not call **_pg_defaultchart,** it must explicitly define every variable in the chart environment—a tedious and unnecessary procedure. The recommended method for adjusting the appearance of your chart is to initialize variables for the proper chart type by calling the

_pg_defaultchart function, and then reassign selected environment variables such as titles.

The following list describes **chartenv** member variables:

Member Variable	Description
chartstyle	An integer that determines the style of the chart (see Table 14.1). Legal values for *chartstyle* are **_PG_PERCENT** and **_PG_NOPERCENT** for pie charts; **_PG_STACKEDBARS** and **_PG_PLAINBARS** for bar and column charts; and **_PG_POINTANDLINE** and **_PG_POINTONLY** for line graphs and scatter diagrams. This variable corresponds to the third argument for the **_pg_defaultchart** function.
charttype	An integer that determines the type of chart displayed. The value of the variable *charttype* is **_PG_BARCHART**, **_PG_COLUMNCHART**, **_PG_LINECHART**, **_PG_SCATTERCHART**, or **_PG_PIECHART**. This variable corresponds to the second argument for the **_pg_defaultchart** function.
chartwindow	A **windowtype** structure that defines the appearance of the chart window.
datawindow	A **windowtype** structure that defines the appearance of the data window.
legend	A **legendtype** structure that defines the appearance of the legend window.
maintitle	A **titletype** structure that defines the appearance of the main title of the chart.
subtitle	A **titletype** structure that defines the appearance of the chart's subtitle.
xaxis	An **axistype** structure that defines the appearance of the *x* axis. (This variable is not applicable for pie charts.)
yaxis	An **axistype** structure that defines the appearance of the *y* axis. (This variable is not applicable for pie charts.)

An Overview of the Presentation Graphics Functions

The chapter concludes with a few words about the 22 functions that make up the Presentation Graphics library. They are listed in Table 14.3 for convenient reference. Refer to Appendix B, "C Library Guide," or online help for a description of the functions and their arguments.

Table 14.3 Presentation Graphics Functions

Primary Functions	Secondary Functions	
_pg_initchart	**_pg_hlabelchart**	**_pg_setpalette**
_pg_defaultchart	**_pg_vlabelchart**	**_pg_resetpalette**
_pg_chart	**_pg_analyzechart**	**_pg_getstyleset**
_pg_chartms	**_pg_analyzechartms**	**_pg_setstyleset**
_pg_chartscatter	**_pg_analyzescatter**	**_pg_resetstyleset**
_pg_chartscatterms	**_pg_analyzescatterms**	**_pg_getchardef**
_pg_chartpie	**_pg_analyzepie**	**_pg_setchardef**
	_pg_getpalette	

In most cases you need only be concerned with seven of the routines, called the "primary functions." These functions initialize variables and display the selected chart types. As demonstrated in example programs earlier in this chapter, you can create very acceptable charts with programs that call only three of the Presentation Graphics primary functions.

The 15 secondary functions of Presentation Graphics do not directly display charts. Most of them retrieve or set data in the Presentation Graphics chart environment.

Of special interest among the secondary functions are the "analysis functions," identified by the prefix **_pg_analyze** in their function names. These five functions calculate default values that pertain to a given chart type and data set. Calling an analysis function has the same effect as calling a corresponding primary function, except that the chart is not displayed. This allows you to pass on to the library the burden of calculating values. You can then make modifications to the resulting values and call a primary routine to display the chart.

Use the **_pg_hlabelchart** and **_pg_vlabelchart** functions to display text on your chart that is not part of a title or axis label. These functions enable you to attach notes or other messages to your chart. You may also find them useful for labeling separate lines of a multiseries line graph.

CHAPTER 15

Fonts

Preceding chapters have discussed how to write QuickC programs that generate graphics and display charts. QuickC has yet another capability when it comes to graphics: fonted text. Your programs can display various styles and sizes of text in any graphics image or chart.

This chapter tells how. It assumes you have already read Chapter 13, "Graphics." You should understand such terms as "graphics mode" and "text mode," and be familiar with the functions **_setvideomode** and **_moveto**. Other than that, there's very little to it. Fonts are simple to learn and even simpler to use, yet they can add to your graphics a final touch of polish.

QuickC Fonts

A "font" is a collection of stylized text characters. Each font consists of several type sizes and a typeface.

"Typeface" is a printer's term that refers to the style of the displayed text—Courier, for example, or Roman. The list on the following page shows six of the typefaces available with QuickC's font library.

"Type size" measures the screen area occupied by individual characters. This term is also borrowed from the printer's lexicon, but for our purposes is specified in units of screen pixels. For example, "Courier 16 × 9" denotes text of Courier typeface, with each character occupying a screen area of 16 vertical pixels by 9 horizontal pixels.

QuickC's font functions use two methods to create fonts. The first technique generates Courier, Helv, and Tms Rmn fonts through a "bit-mapping" (or "raster-mapping") technique. Bit-mapping defines character images with binary data. Each bit in the map corresponds to a screen pixel. If a bit is 1, its associated pixel is set to the current screen color. A bit value of 0 clears the pixel. Video adapters use this same technique to display text in graphics mode.

The second method creates the remaining three type styles—Modern, Script, and Roman—as "vector-mapped" fonts. Vector-mapping represents each character in terms of lines and arcs. In a literal sense vector-mapped characters are drawn on the screen. You might think of bit-mapped characters as being stenciled.

Each method has advantages and disadvantages. Bit-mapped characters are more completely formed since the pixel mapping is predetermined. However, they cannot be scaled. Vector-mapped text can be scaled to any size, but the characters tend to lack the solid appearance of the bit-mapped characters.

Typeface	Sample Text
Courier	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Helv	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Tms Rmn	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Modern	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Script	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Roman	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz

Table 15.1 lists available sizes for each font. Notice that the bit-mapped fonts come in preset sizes as measured in pixels. The exact size of any fonted character depends on screen resolution and display type.

Table 15.1 Typefaces and Type Sizes in the QuickC Library

Typeface	Mapping	Size (in pixels)	Spacing
Courier	Bit	13 × 8, 16 × 9, 20 × 12	Fixed
Helv	Bit	13 × 5, 16 × 7, 20 × 8, 13 × 15, 16 × 6, 19 × 8	Fixed
Tms Rmn	Bit	10 × 5, 12 × 6, 15 × 8, 16 × 9, 20 × 12, 26 × 16	Fixed
Modern	Vector	Scaled	Proportional
Script	Vector	Scaled	Proportional
Roman	Vector	Scaled	Proportional

QuickC's font routines can display characters 32–255, including most extended characters (ASCII 128–255). A few extended characters cannot be displayed; these are represented as either an underscore (_) or period (.) character.

Using QuickC's Font Library

Data for both bit-mapped and vector-mapped fonts reside in files on disk. A .FON extension identifies the files. The names of the .FON files indicate their content. For example, the files MODERN.FON, ROMAN.FON, and SCRIPT.FON hold data for the three vector-mapped fonts.

You can use Microsoft Windows .FON files.

QuickC .FON files are identical to the .FON files used in the Microsoft Windows operating environment. If you have access to Windows you can use any of its .FON files with QuickC's font functions. Windows .FON files are also available for purchase separately. In addition, several vendors offer software that can create or modify .FON files, allowing you to design your own fonts.

Your programs should follow these three steps to display fonted text:

1. Register fonts
2. Set the current font from the register
3. Display text using the current font

The following sections describe each of the three steps in detail. An example program later in the chapter demonstrates the steps.

Register Fonts

The fonts you plan to use must first be organized into a list in memory, a process called "registering." The register list contains information about the available .FON files. Register fonts by calling the function **_registerfonts**. This function reads header information from specified .FON files. It builds a list of file information but does not read mapping data from the files.

The GRAPH.H file prototypes the **_registerfonts** function as:

```
short far _registerfonts( unsigned char far * );
```

The argument points to a string containing a file name. The file name is the name of the .FON file for the desired font. The file name can include wild cards, allowing you to register several fonts with one call to **_registerfonts**.

If it successfully reads one or more .FON files, **_registerfonts** returns the number of fonts registered. If the function fails, it returns a negative error code. Refer to Appendix B, "C Library Guide," or to online help for a description of error codes.

Set Current Font

Call the function **_setfont** to select a current font. This function checks to see if the requested font is registered, then reads the mapping data from the appropriate .FON file. A font must be registered and marked current before your program can display text of that font.

The GRAPH.H file prototypes **_setfont** as

```
short far _setfont( unsigned char far * );
```

The function's argument is a pointer to a character string. The string consists of letter codes that describe the desired font, as outlined below:

Option Code	Meaning
b	Select the best fit from the registered fonts. This option instructs **_setfont** to accept the closest-fitting font if a font of the specified size is not registered.

If at least one font is registered, the **b** option always sets a current font. If you do not specify the **b** option and an exact matching font is not registered, **_setfont** will fail. In this case, any existing current font remains current. Refer to online help for a description of error codes returned by **_setfont**.

The **_setfont** function uses four criteria for selecting the best fit. In descending order of precedence the four criteria are pixel height, typeface, pixel width, and spacing (fixed or proportional). If you request a vector-mapped font, **_setfont** sizes the font to correspond with the specified pixel height and width. If you request a raster-mapped (bit-mapped) font, **_setfont** chooses the closest available size. If the requested type size for a raster-mapped font fits exactly between two registered fonts, the smaller size takes precedence.

f — Select only a fixed-spaced font.

h*y* — Character height, where *y* is the height in pixels.

n*x* — Select font number *x*, where *x* is less than or equal to the value returned by **_registerfonts**. For example, the option **n3** makes the third registered font current, assuming that three or more fonts are registered.

p — Select only a proportional-spaced font.

r — Select only a raster-mapped (bit-mapped) font.

t'*fontname*' — Typeface of the font in single quotes. The *fontname* string is one of the following:

courier	modern
helv	script
tms rmn	roman

Notice the space in "tms rmn." Additional font files use other names for *fontname*. Refer to the vendor's documentation for these names.

v — Select only a vector-mapped font.

w*x* — Character width, where *x* is the width in pixels.

Option codes are not case-sensitive and can be listed in any order. You can separate codes with spaces or any other character that is not a valid option code. The **_setfont** function ignores all invalid codes.

The **_setfont** function updates a data area with parameters of the current font. The data area is in the form of a structure, defined in the GRAPH.H file as

```
struct _fontinfo
{
   int     type;          /* set = vector,clear = bit map */
   int     ascent;        /* pix dist from top to base */
   int     pixwidth;      /* character width in pixels */
   int     pixheight;     /* character height in pixels */
   int     avgwidth;      /* average character width */
   char    filename[81];  /* file name including path */
   char    faceName[32];  /* font name */
};
```

If you wish to retrieve the parameters of the current font, call the function **_getfontinfo**. Refer to Appendix B, "C Library Guide," or online help for a description of this function.

Display Text

The last step consists of two parts. First, select a screen position for the text with the graphics function **_moveto**. Then display fonted text at that position with the function **_outgtext**. The **_moveto** function takes pixel coordinates as arguments. The coordinates locate the top left of the first character in the text string.

An Example Program

QuickC's font functions shine when used in conjunction with your other graphics functions. They allow you to dress up any image on the screen. Yet they can make a visual impression when used by themselves, as an example will show.

The program SAMPLER.C displays sample text in all the available fonts, then exits when a key is pressed. Make sure the .FON files are in the current directory before running the program.

```
/* SAMPLER.C: Display sample text in various fonts. */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
#include <string.h>
```

```
#define NFONTS 6

main()

{
   static unsigned char *text[2*NFONTS] =
   {
       "COURIER",        "courier",
       "HELV",           "helv",
       "TMS RMN",        "tms rmn",
       "MODERN",         "modern",
       "SCRIPT",         "script",
       "ROMAN",          "roman"
   };
   static unsigned char *face[NFONTS] =
   {
       "t'courier'",
       "t'helv'",
       "t'tms rmn'",
       "t'modern'",
       "t'script'",
       "t'roman'"
   };
   static unsigned char list[20];
   struct videoconfig vc;
   int mode = _VRES16COLOR;
   register i;

   /*   Read header info from all .FON files in
    *   current directory    */

   if( _registerfonts( "*.FON" ) < 0 )
   {
      _outtext( "Error:  can't register fonts" );
      exit( 0 );
   }

   /*   Set highest available video mode */

   while( !_setvideomode( mode ) )
      mode--;
   if( mode == _TEXTMONO )
      exit ( 0 );

   /*   Copy video configuration into structure vc */

   _getvideoconfig( &vc );
```

```
/*   Display six lines of sample text */

for( i = 0; i < NFONTS; i++ )
{
   strcpy( list, face[i] );
   strcat( list, "h30w24b" );

*  if( !_setfont( list ) )
   {
       _setcolor( i + 1 );
       _moveto( 0, (i * vc.numypixels) / NFONTS );
       _outgtext( text[i * 2] );
       _moveto( vc.numxpixels / 2,
                (i * vc.numypixels) / NFONTS );
       _outgtext( text[(i * 2) + 1] );
   }
   else
   {
       _setvideomode( _DEFAULTMODE );
       _outtext( "Error:  can't set font" );
       exit( 0 );
   }
}
getch();
_setvideomode( _DEFAULTMODE );

/* Return memory when finished with fonts */

_unregisterfonts();
exit ( 0 );
}
```

Notice that SAMPLER.C calls the graphics function **_moveto** to establish the starting position for each text string. Chapter 13, "Graphics," describes the **_moveto** function in the section "Graphics Coordinates." The function **_setfont** takes a character string as an argument. The string is a list of options that specifies typeface and the best fit for a character height of 30 pixels, and a width of 24 pixels. See Appendix B, "C Library Guide," and online help for complete descriptions of the QuickC font functions.

A Few Hints

Fonted text is simply another form of graphics, and using fonts effectively requires little programming effort. Still, there are a few things to watch:

- Remember the video should be set only once to establish a graphics mode. If you generate an image—say, with Presentation Graphics—and wish to incorporate fonted text into it, don't reset the video mode prior to calling the font routines. Doing so will blank the screen, destroying the original image.

* **Errata:** `if( _setfont( list ) >= 0 )`

- The _setfont function reads specified .FON files to obtain mapping data for the current font. Each call to _setfont causes a disk access and overwrites the old font data in memory. If you wish to show text of different styles on the same screen, display all text of one font before moving on to the others. By minimizing the number of calls to _setfont you'll save time spent in disk I/O and memory reloads.
- When your program finishes with the fonts library, you might wish to free the memory occupied by the register list. Call the function _unregisterfonts to do this. As its name implies, this function frees the memory allocated by _registerfonts. The register information for each type size of each font takes up approximately 140 bytes of memory. Thus the amount of memory returned by _unregisterfonts is significant only if you have many fonts registered.
- As for aesthetics, the same suggestions for the printed page apply to fonted screen text. Typefaces are more effective when they are not competing with each other for attention. Restricting the number of styles per screen to one or two generally results in a more pleasing, less cluttered image.

CHAPTER 16

In-Line Assembly

QuickC has the ability to handle assembly-language instructions right in your C programs. This powerful feature is called "in-line assembly."

Assembly language serves many purposes, such as improving program speed, reducing memory needs, and controlling hardware. The in-line assembler lets you embed assembly-language instructions directly in your C source programs without extra assembly and link steps. And the assembler is built into the compiler—you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

This chapter assumes that you are familiar with assembly-language terms and concepts. If you have never programmed in assembly language, refer to the section "References and Books on Assembly Language," at the end of this chapter.

Advantages of In-Line Assembly

Because QuickC's in-line assembler doesn't require separate assembly and link steps, it is more convenient than a separate assembler. In-line assembly code can use any C variable or function name that is visible (in scope), so it is easy to integrate it with your program's C code. And because the assembly code can be mixed in-line with C statements, it can do tasks that are cumbersome or impossible in C alone.

The uses of in-line assembly include

- Writing the body of a function in assembly language
- Spot-optimizing speed-critical sections of code
- Calling DOS and BIOS routines with the **INT** instruction
- Creating TSR (terminate-and-stay-resident) code or handler routines that require knowledge of processor states

In-line assembly is a special-purpose tool. If you plan to transport an application, you'll probably want to place machine-specific code in a separate module. And because the in-line assembler doesn't support all MASM directives, you may find it more convenient to use MASM for such modules.

The _asm Keyword

The **_asm** keyword invokes the in-line assembler and can appear wherever a C statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces. The term "**_asm** block" here refers to any instruction or group of instructions, whether or not in braces.

Below is a simple **_asm** block enclosed in braces. (The code prints the "beep" character, ASCII 7.)

```
_asm
{
   mov ah, 2
   mov dl, 7
   int 21h
}
```

Alternatively, you can put **_asm** in front of each assembly instruction:

```
_asm mov ah, 2
_asm mov dl, 7
_asm int 21h
```

Since the **_asm** keyword is a statement separator, you can also put assembly instructions on the same line:

```
_asm mov ah, 2   _asm mov dl, 7   _asm int 21h
```

Braces can prevent ambiguity and needless repetition.

All three examples generate the same code, but the first style—enclosing the **_asm** block in braces—has some advantages. The braces clearly separate assembly code from C code and avoid needless repetition of the **_asm** keyword. Braces can also prevent ambiguities. If you want to put a C statement on the same line as an **_asm** block, you must enclose the block in braces. Without the braces, the compiler cannot tell where assembly code stops and C statements begin. Finally, since the text in braces has the same format as ordinary MASM text, you can easily cut and paste text from existing MASM source files.

The braces enclosing an **_asm** block don't affect variable visibility, as do braces in C. You can also nest **_asm** blocks, but the nesting doesn't affect variable visibility.

Using Assembly Language in _asm Blocks

The in-line assembler has much in common with other assemblers. For example, it accepts any expression that is legal in MASM, and it supports almost all 80286 and 80287 instructions. This section describes the use of assembly-language features in **_asm** blocks.

Instruction Set

The in-line assembler supports the full instruction set of the Intel® 80286 and 80287 processors, except for privileged instructions that control the processor's protected mode (protected mode is available in the OS/2 and XENIX® operating systems, but not in DOS). It does not recognize 80386- and 80387-specific instructions. To use assembly instructions specific to the 80286 and 80287 processors, you must compile your QuickC program with the /G2 switch included in the command line. For a description of the compiler /G command-line switch, refer to Chapter 4, "QCL Command Reference," in the *Microsoft QuickC Tool Kit.*

Expressions

In-line assembly code can use any MASM expression, that is, any combination of operands and operators that evaluates to a single value or address.

Data Directives and Operators

Although an _asm block can reference C data types and objects, it cannot define data objects with MASM directives or operators. Specifically, you cannot use the definition directives **DB, DW, DD, DQ, DT,** and **DF**, or the operators **DUP** or **THIS**. Nor are MASM structures and records available. The in-line assembler doesn't accept the directives **STRUC, RECORD, WIDTH,** or **MASK.**

EVEN and ALIGN Directives

While the in-line assembler doesn't support most MASM directives, it does support **EVEN** and **ALIGN**. These directives put **NOP** (no operation) instructions in the assembly code as needed to align labels to specific boundaries. This makes instruction-fetch operations more efficient for some processors (not including eight-bit processors such as the Intel 8088).

Macros

The in-line assembler is not a macro assembler. You cannot use MASM macro directives (**MACRO, REPT, IRC, IRP,** and **ENDM**) or macro operators (**<>, !, &,** **%**, and **.TYPE**). An **_asm** block can use C preprocessor directives, however. See the section "Using C in **_asm** Blocks" for more information.

Segment References

You must refer to segments by register rather than by name (the segment name **_TEXT** is invalid, for instance). Segment overrides must use the register explicitly, as in ES:[BX].

Type and Variable Sizes

The **LENGTH, SIZE,** and **TYPE** operators have a limited meaning in in-line assembly. They cannot be used at all with the **DUP** operator (because you cannot define data with MASM directives or operators). But you can use them to find the size of C variables or types:

- The **LENGTH** operator can return the number of elements in an array. It returns the value 1 for nonarray variables.
- The **SIZE** operator can return the size of a C variable. A variable's size is the product of its **LENGTH** and **TYPE.**
- The **TYPE** operator can return the size of a C type or variable. If the variable is an array, **TYPE** returns the size of a single element of the array.

For instance, if your program has an eight-element **int** array,

```
int arr[8];
```

the following C and assembly expressions yield the size of `arr` and its elements:

_asm	C	Size
`LENGTH arr`	`sizeof(arr)/sizeof(arr[0])`	8
`SIZE arr`	`sizeof(arr)`	16
`TYPE arr`	`sizeof(arr[0])`	2

Comments

Instructions in an **_asm** block can use assembly-language comments:

```
_asm mov ax, offset buff ; Load address of buff
```

Because C macros expand into a single logical line, avoid using assembly-language comments in macros (see the section "Defining **_asm** Blocks as C Macros," below). An **_asm** block can also contain C-style comments, as noted below.

Debugging with the CodeView® Debugger

In-line assembly code can be debugged with CodeView.

Programs containing in-line assembly code can be debugged with the CodeView debugger, assuming you compile with the /Zi option.

Note that putting multiple assembly instructions or C statements on one line can hamper debugging with CodeView. In source mode, the CodeView debugger lets you set breakpoints on a single line but not on individual statements on the same line. The same principle applies to an **_asm** block defined as a C macro, which expands to a single logical line.

Using C in _asm Blocks

Because in-line assembly instructions can be mixed with C statements, they can refer to C variables by name and use many other elements of C. An **_asm** block can use the following C language elements:

- Symbols, including labels and variable and function names
- Constants, including symbolic constants and **enum** members

- Macros and preprocessor directives
- Comments (/* */)
- Type names (wherever a MASM type would be legal)
- **typedef** names, generally used with operators such as **PTR** and **TYPE** or to specify structure or union members

Within an **_asm** block, you can specify integer constants with either C notation or assembler radix notation (0x100 and 100h are equivalent, for instance). This allows you to define (using **#define**) a constant in C, and use it in both C and assembly portions of the program. You can also specify constants in octal by preceding them with a 0. For example, 0777 specifies an octal constant.

Using Operators

An **_asm** block cannot use C-specific operators, such as the << operator. However, operators shared by QuickC and MASM, such as the * operator, are interpreted as assembly-language operators. For instance, outside an **_asm** block, square brackets ([]) are interpreted as enclosing array subscripts, which C automatically scales to the size of an element in the array. Inside an **_asm** block, they are seen as the MASM index operator, which yields an unscaled byte offset from any data object or label (not just an array). The following code illustrates the difference:

```
int array[10];

_asm mov array[6], bx ;  Store BX at array+6 (not scaled)

array[6] = 0;         /* Store 0 at array+12 (scaled) */
```

The first reference to `array` is not scaled, but the second is. Note that you can use the **TYPE** operator to achieve scaling based on a constant. For instance, the following statements are equivalent:

```
_asm mov array[6 * TYPE int], 0 ; Store 0 at array + 12

array[6] = 0;                   /* Store 0 at array + 12 */
```

Using C Symbols

An **_asm** block can refer to any C symbol that is visible (in scope) where the block appears. (C symbols are variable names, function names, and labels—in other words, names that aren't symbolic constants or **enum** members.)

A few restrictions apply to the use of C symbols:

- Each assembly-language statement can contain only one C symbol. Multiple symbols can appear in the same assembly instruction only with **OFFSET**, **LENGTH**, **TYPE**, and **SIZE** expressions.
- Functions referenced in an **_asm** block must be declared (prototyped) earlier in the program. Otherwise, the compiler cannot distinguish between function names and labels in the **_asm** block.
- An **_asm** block cannot use any C symbols with the same spelling as MASM reserved words (regardless of case). MASM reserved words include instruction names such as **PUSH** and register names such as SI.
- Structure and union tags are not recognized in **_asm** blocks.

Accessing C Data

A great convenience of in-line assembly is the ability to refer to C variables by name. An **_asm** block can refer to any symbols—including variable names—that are visible where the block appears. For instance, if the C variable `var` is visible, the instruction

```
_asm mov ax, var
```

stores the value of `var` in AX.

If a structure or union member has a unique name, an **_asm** block can refer to it using only the member name, without specifying the C variable or **typedef** name before the period (.) operator. If the member name is not unique, however, you must place a variable or **typedef** name immediately before the period (.) operator. For instance, the following structure types share `same_name` as their member name:

```
struct first_type
{
   char *weasel;
   int same_name;
};

struct second_type
{
   int wonton;
   long same_name;
};
```

If you declare variables with the types

```
struct first_type hal;
struct second_type oat;
```

all references to the member `same_name` must use the variable name, because `same_name` is not unique. But the member `weasel` has a unique name, so you can refer to it using only its member name:

```
_asm
{
   mov bx, OFFSET hal
   mov cx, [bx]hal.same_name ; Must use 'hal'
   mov si, [bx].weasel       ; Can omit 'hal'
}
```

Note that omitting the variable name is merely a coding convenience. The same assembly instructions are generated whether or not it is present.

Writing Functions

If you write a function with in-line assembly code, it's a simple matter to pass arguments to the function and return a value from it. The following examples compare a function first written for a separate assembler and then rewritten for the in-line assembler. The function, called `power2`, receives two parameters, multiplying the first parameter by 2 to the power of the second parameter. Written for a separate assembler, the function might look like this:

```
; POWER.ASM
; Compute the power of an integer
;
       PUBLIC _power2
_TEXT SEGMENT WORD PUBLIC 'CODE'
_power2 PROC

        push bp          ; Save BP
        mov bp, sp       ; Move SP into BP so we can refer
                         ;   to arguments on the stack
        mov ax, [bp+4]   ; Get first argument
        mov cx, [bp+6]   ; Get second argument
        shl ax, cl       ; AX = AX * ( 2 ^ CL )
        pop bp           ; Restore BP
        ret              ; Return with sum in AX

_power2 ENDP
_TEXT   ENDS
        END
```

Function arguments are usually passed on the stack.

Since it's written for a separate assembler, the function requires a separate source file and assembly and link steps. C function arguments usually are passed on the stack, so this version of the `power2` function accesses its arguments by their positions on the stack. (Note that the **MODEL** directive, available in MASM and some other assemblers, also allows you to access stack arguments and local stack variables by name.)

The POWER2.C program below writes the `power2` function with in-line assembly code:

```
/* POWER2.C */
#include <stdio.h>

int power2( int num, int power );

void main( void )
{
   printf( "3 times 2 to the power of 5 is %d\n", \
           power2( 3, 5) );
}

int power2( int num, int power )
{
   _asm
   {
      mov ax, num    ; Get first argument
      mov cx, power  ; Get second argument
      shl ax, cl     ; AX = AX * ( 2 to the power of CL )
   }
   /* Return with result in AX */
}
```

The in-line version of the `power2` function refers to its arguments by name and appears in the same source file as the rest of the program. This version also requires fewer assembly instructions. Since C automatically preserves BP, the **_asm** block doesn't need to do so. It can also dispense with the **RET** instruction, since the C part of the function performs the return.

Because the in-line version of `power2` doesn't execute a C **return** statement, it causes a harmless warning if you compile at warning levels 2 or higher:

```
warning C4035: 'power2' : no return value
```

The function does return a value, but QuickC cannot tell that in the absence of a **return** statement. Simply ignore the warning in this context.

Using and Preserving Registers

In general, you should not assume that a register will have a given value when an **_asm** block begins. An **_asm** block inherits whatever register values happen to result from the normal flow of control.

As you may have noticed in the POWER2.C example in the previous section, the `power2` function doesn't preserve the value in the AX register. When you write a function in assembly language, you don't need to preserve the AX, BX, CX, DX, ES, and flags registers. However, you should preserve any other registers you use (DI, SI, DS, SS, SP, and BP).

WARNING *If your in-line assembly code changes the direction flag using the STD or CLD instructions, you must restore the flag to its original value.*

The POWER2.C example in the previous section also shows that functions return values in registers. This is true whether the function is written in assembly language or in C.

Functions return values in the AX and DX registers.

If the return value is short (a **char**, **int**, or **near** pointer), it is stored in AX. The POWER2.C example returned a value by terminating with the desired value in AX.

If the return value is long, store the high word in DX and the low word in AX. To return a longer value (such as a floating-point value), store the value in memory and return a pointer to the value (in AX if **near** or in DX:AX if **far**).

Assembly instructions that appear in-line with C statements are free to alter the AX, BX, CX, and DX registers. C doesn't expect these registers to be maintained between statements, so you don't need to preserve them. The same is true of the SI and DI registers, with some exceptions (see the section "Optimizing," below). You should preserve the SP and BP registers unless you have some reason to change them—to switch stacks, for instance.

Jumping to Labels

Like an ordinary C label, a label in an **_asm** block is visible (has scope) throughout the function in which it is defined (not only in the block). Both assembly instructions and C **goto** statements can jump to labels inside or outside the **_asm** block.

Labels in _asm blocks have function scope and are not case sensitive.

Unlike C labels, labels defined in **_asm** blocks are not case sensitive, even when used in C statements. C labels are not case sensitive in an **_asm** block, either. (Outside an **_asm** block, a C label is case sensitive as usual.) The following do-nothing code shows all the permutations.

```
void func( void )
{
   goto C_Dest;  /* legal */
   goto c_dest;  /* error */

   goto A_Dest;  /* legal */
   goto a_dest;  /* legal */

   _asm
   {
      jmp C_Dest ; legal
      jmp c_dest ; legal

      jmp A_Dest ; legal
      jmp a_dest ; legal

      a_dest:    ; _asm label
   }

   C_Dest:       /* C label */
   return;
}
```

Don't use C library function names as labels in **_asm** blocks. For instance, you might be tempted to use `exit` as a label,

```
jne exit
   .
   .
   .
exit:
   ; More _asm code follows
```

forgetting that **exit** is the name of a C library function. The code doesn't cause a compiler error, but it might cause a jump to the **exit** function instead of the desired location.

As in MASM programs, the dollar symbol ($) serves as the current location counter—a label for the instruction currently being assembled. In **_asm** blocks, its main use is to make long conditional jumps:

```
jne $+5 ; next instruction is 5 bytes long
jmp farlabel
; $+5
   .
   .
   .
farlabel:
```

Calling C Functions

An **_asm** block can call C functions, including C library routines. The following example calls the **printf** library routine:

```
#include <stdio.h>

char format[] = "%s %s\n";
char hello[] = "Hello";
char world[] = "world";

void main( void )
{
   _asm
   {
      mov  ax, offset world
      push ax
      mov  ax, offset hello
      push ax
      mov  ax, offset format
      push ax
      call printf
      add sp, 6
   }
}
```

Since function arguments are passed on the stack, you simply push the needed arguments—string pointers, in the example above—before calling the function. The arguments are pushed in reverse order, so they come off the stack in the desired order. To emulate the C statement

```
printf( format, hello, world );
```

the example pushes pointers to `world`, `hello`, and `format`, in that order, then calls **printf**. The last instruction in the **_asm** block adjusts the stack to account for the arguments previously pushed onto it.

Defining _asm Blocks as C Macros

C macros offer a convenient way to insert assembly code into C code, but they demand extra care because a macro expands into a single logical line. To create trouble-free macros, follow these rules:

- Enclose the _**asm** block in braces
- Put the _**asm** keyword in front of each assembly instruction
- Use old-style C comments (`/* comment */`) instead of assembly-style comments (`; comment`)

To illustrate, the following example defines a simple macro:

```
#define BEEP _asm \
/* Beep sound */      \
{                     \
   _asm mov ah, 2     \
   _asm mov dl, 7     \
   _asm int 21h       \
}
```

At first glance, the last three _**asm** keywords seem superfluous. They are needed, however, because the macro expands into a single line:

```
_asm /* Beep sound */ { _asm mov ah, 2  _asm mov dl, 7 _asm int 21h }
```

The third and fourth _**asm** keywords are needed as statement separators. The only statement separators recognized in _**asm** blocks are the newline character and _**asm** keyword. And since a block defined as a macro is one logical line, you must separate each instruction with _**asm**.

The braces are essential as well. If you omit them, the compiler can be confused by C statements on the same line to the right of the macro invocation. Without the closing brace, QuickC cannot tell where assembly code stops, and it sees C statements after the _**asm** block as assembly instructions.

Use C comments in _asm blocks written as macros.

Assembly-style comments that start with a semicolon (;) continue to the end of the line. This causes problems in macros because QuickC ignores everything after the comment, all the way to the end of the logical line. To prevent errors, use C comments (`/* comment */`) in _**asm** blocks defined as macros.

An _asm block written as a C macro can take arguments. Unlike an ordinary C macro, however, an _asm macro cannot return a value. So you cannot use such macros in C expressions.

You can convert MASM macros to C macros.

Note that some MASM-style macros can be written as C macros. Below is a MASM macro that sets the video page to the value specified in the `page` argument:

```
setpage   MACRO page
          mov ah, 5
          mov al, page
          int 10h
          ENDM
```

The following code defines `setpage` as a C macro:

```
#define setpage( page ) _asm  \
   {                           \
      _asm mov ah, 5           \
      _asm mov al, page        \
      _asm int 10h             \
}
```

Both macros do the same job.

Optimizing

The presence of an _asm block in a function affects optimization in a few different ways. First, as you might expect, QuickC doesn't try to optimize the _asm block itself. What you write in assembly language is exactly what you get.

Second, the presence of an _asm block affects register variable storage. (See the section "Register Variables" in Chapter 5, "Advanced Data Types," for a discussion of register variables.) Under normal circumstances, QuickC automatically stores variables in registers. This is not done, however, in any function that contains an _asm block. To get register variable storage in such a function, you must request it with the **register** keyword.

Since the compiler stores register variables in the SI and DI registers, these registers represent variables in functions that request register storage. The first eligible variable is stored in SI and the second in DI. Preserve SI and DI in such functions unless you want to change the register variables.

Keep in mind that the name of a variable declared with **register** translates directly into a register reference (assuming a register is available for such use). For instance, if you declare

```
register int sample;
```

and the variable `sample` happens to be stored in SI, then the **_asm** instruction

```
_asm mov ax, sample
```

is equivalent to

```
_asm mov ax, si
```

If you declare a variable with **register** and the compiler cannot store the variable in a register, QuickC issues a compiler error if you reference the variable in an **_asm** block. The solution is to remove the **register** declaration from that variable.

Register variables form a slight exception to the general rule that an assembly-language statement can contain no more than one C symbol. If one of the symbols is a register variable, for example,

```
register int v1;
int v2;
```

then an instruction can use two C symbols, as in

```
mov v1, v2
```

Finally, the presence of in-line assembly code inhibits loop optimization for the entire function in which the code appears. (Loop optimization can be selected with the /Ol command-line switch; see Chapter 4, "QCL Command Reference," in *Microsoft QuickC Tool Kit.*) This optimization is suppressed no matter which compiler options you use.

References and Books on Assembly Language

Assembly language varies widely for different computer processors. In selecting a reference on assembly language, make sure it describes assembly for the Intel 8086 family of processors or compatibles. These are the microprocessors used in the IBM and IBM-compatible computers able to run QuickC.

The following books and articles may be useful in learning to program in assembly language:

Chesley, Harry R. and Mitchell Waite. *Supercharging C with Assembly Language*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1987.

Duncan, Ray. *Advanced MS-DOS Programming*, 2nd ed. Redmond, Washington: Microsoft Press, 1988.

Lafore, Robert. *Assembly Language Primer for the IBM PC & XT*. New York, New York: Plume/Waite, 1984.

Metcalf, Christopher D. and Marc B. Sugiyama. *COMPUTE!'s Beginner's Guide to Machine Language on the IBM PC & PCjr*. Greensboro, North Carolina: COMPUTE! Publications, Inc., 1985.

Microsoft. *Microsoft Macro Assembler 5.1 Programmer's Guide*. Redmond, Washington, 1987. (Included with Microsoft Macro Assembler.)

Microsoft. *Microsoft Macro Assembler 5.1 Reference*. Redmond, Washington, 1987. (Included with Microsoft Macro Assembler.)

Sargent, Murray and Richard L. Shoemaker. *The IBM Personal Computer from the Inside Out*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1986.

The above references are listed for your convenience only. With the exception of those published by Microsoft, Microsoft Corporation does not endorse these books or recommend them over others on the same subject.

Appendixes

Appendix A
C Language Guide

This appendix provides a quick summary of C language fundamentals. It does not attempt to teach you the C language (Part 1 of this book does that) or document all the details of C. Use it as a refresher or ready reference after you have read all the material in Chapters 1 through 10.

To simplify reference, this appendix has the same general organization as the chapters in Part 1. Each major section lists the chapter(s) where you may find more detailed information on a given topic.

You can also use QuickC's online help to get instant information on any topic. The online help index and table of contents provide alternate ways to access information.

General Syntax

Basic C-language syntax is explained in Chapter 1, "Anatomy of a C Program."

A C statement consists of keywords, expressions, and function calls. A statement always ends with a semicolon. A statement block is a collection of statements enclosed by braces ({ }). A statement block can appear anywhere a simple C statement appears. No semicolon occurs after the closing brace.

C is a free-format programming language. You can insert "whitespace" characters (spaces, tabs, carriage returns, and form feeds) almost anywhere, to indent statement blocks and otherwise make your code more readable.

Comments begin with the slash-asterisk sequence (/*) and end with the asterisk-slash sequence (*/). Comments are legal anywhere a space is legal, but they cannot be nested.

User-Defined Names

The rules governing user-defined names are explained in Chapter 1, "Anatomy of a C Program," and Chapter 4, "Basic Data Types."

You can define your own names ("identifiers") for variables, functions, and user-defined types. Identifiers are case sensitive. For instance, the identifier `myVariable` is not the same as the identifier `Myvariable`. You cannot use a C keyword (see the list below) as an identifier.

An identifier can contain only the following characters:

- abcdefghijklmnopqrstuvwxyz
- ABCDEFGHIJKLMNOPQRSTUVWXYZ
- 0123456789
- _ (underscore)

The first character of an identifier must be a letter or the underscore character. The first 31 characters of local identifiers are significant. The name can contain more than 31 characters, but QuickC ignores everything beyond the thirty-first character. Global identifiers are normally significant to 30 characters.

Keywords

A keyword has a special meaning in the C language. You must spell keywords as shown in the following list, and you cannot use them as user-defined names (see above).

_asm	**_emit**	**_interrupt**	**sizeof**
auto	**enum**	**_loadds**	**static**
_based	**_export**	**long**	**struct**
break	**extern**	**_near**	**switch**
case	**_far**	**_pascal**	**typedef**
_cdecl	**_fastcall**	**register**	**union**
char	**float**	**return**	**unsigned**
const	**for**	**_saveregs**	**void**
continue	**_fortran**	**_segname**	**volatile**
default	**goto**	**_segment**	**while**
do	**_huge**	**_self**	
double	**if**	**short**	
else	**int**	**signed**	

A few other words, such as **main**, have a special meaning but are not keywords in the strict sense. Use online help to get details on all such words.

Functions

The rules governing C functions are explained in Chapter 2, "Functions."

Every C program must have at least one function, named **main**, which marks the beginning and end of the program's execution. Every executable statement in a C program must occur within a function.

Variables can be declared inside or outside functions. Variables declared inside a function are "local" and can only be accessed in that function. Variables declared outside all functions are "global" and can be accessed from any function in your program.

You call a C function by stating its name. If the function requires "arguments" (data), you list the arguments in the parentheses that follow the function name. Arguments that you pass to a function become local variables in the function.

A function can return a value (using the **return** keyword) or return nothing. If the function contains no **return** statement, it ends automatically when execution reaches the closing brace of the function definition.

A function "prototype" (declaration) tells QuickC the function's name, the type of value it returns, and the number and type of arguments it requires. Function prototypes normally appear near the beginning of the program. They allow QuickC to check the accuracy of every reference to the function.

Flow Control

Flow-control statements are explained in Chapter 3, "Flow Control."

The C language provides several kinds of flow-control statements. The **for**, **while**, and **do** statements create loops. The **if** and **switch** statements perform a branch. The **break**, **continue**, **return**, and **goto** statements perform an unconditional "jump" to another location in your program.

The following sections describe the C flow-control statements in alphabetical order.

The break Statement

The **break** statement terminates the smallest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. It passes control to the statement following the terminated statement.

This statement is often used to exit from a loop or **switch** statement (see below). The following example illustrates **break**:

```
while( c != 'Q' )
{
   /* Some C statements here */
   if( number_of_characters > 80 )
      break;  /* Break out of while loop */
   /* More C statements here */
}
/* Execution continues here after break statement */
```

The continue Statement

The **continue** statement is the opposite of the **break** statement. It passes control to the next iteration of the smallest enclosing **do**, **for**, or **while** statement in which it appears.

This statement is often used to return to the start of a loop from within a deeply nested loop.

The following example illustrates **continue**:

```
while( c != 'Q' )
{
   /* Some C statements here*/
   if( c == 0x20 )
      continue;     /* Skip rest of loop */
   /* More C statements here */
}
```

In the example, the **continue** statement skips to the next iteration of the loop whenever `c` equals 0x20, the ASCII value for a space character.

The do Statement

The **do** statement repeats a statement until a specified expression becomes false. The test expression in the loop is evaluated after the body of the loop executes. Thus, the body of a **do** loop always executes at least once.

Use a **break**, **goto**, or **return** statement when you need to exit a **do** loop early. Use the **continue** statement to terminate an iteration without exiting the loop. The **continue** statement passes control to the next iteration of the loop.

The following example illustrates **do**:

```
sample = 1;
do
   printf( "%d\t%d\n", sample, sample * sample );
while( ++x <= 7 );
```

The **printf** statement in the example always executes at least once, no matter what value `x` has when the loop begins.

The for Statement

The **for** statement lets you repeat a statement a specified number of times. It consists of three expressions:

- An initializing expression, which is evaluated when the loop begins
- A test expression, which is evaluated before each iteration of the loop

- A modifying expression, which is evaluated at the end of each iteration of the loop

These expressions are enclosed in parentheses and followed by the loop body—the statement the loop is to execute. Each expression in the parentheses can be any legal C statement.

The **for** statement works as follows:

1. The initializing expression is evaluated.
2. As long as the test expression evaluates to a nonzero value, the loop body is executed. When the test expression becomes 0, control passes to the statement following the loop body.
3. At the end of each iteration of the loop, the modifying expression is evaluated.

You can use a **break**, **goto**, or **return** statement to exit a **for** loop early. Use the **continue** statement to terminate an iteration without exiting the **for** loop. The **continue** statement passes control to the next iteration of the loop.

The following example illustrates **for**:

```
for( counter = 0; counter < 100; counter++ )
{
   x[counter] = 0; /* Set every array element to zero */
}
```

The goto Statement

The **goto** statement performs a jump to the statement following the specified label. A **goto** statement can jump anywhere within the current function.

A common use of **goto** is to exit immediately from a deeply nested loop. For instance:

```
for( ... )
{
   for( ... )
   {
      /* Do something here */
      if(c == CTRL_C)
         goto myplace;
   }
   /* Do something else here */
}

/* The goto label is named myplace */
myplace:
/* The goto statement transfers control here */
```

The if Statement

The **if** statement performs a branch based on the outcome of a conditional test. If the test expression is true, the body of the **if** statement executes. If it is false, the statement body is skipped.

The **else** keyword is used with **if** to form an either-or construct that executes one statement when the test expression is true and another when it's false. C does not offer an "else-if" keyword. You can combine **if** and **else** statements to achieve the same effect. C pairs each **else** with the most recent **if** that lacks an **else**.

Below is a simple **if** statement:

```
if( score < 70 )
   grade = 'F';
else
   grade = 'P';
```

If the value of the variable `score` is less than 70, the variable `grade` is set to the constant `F`. Otherwise, `score` is set to `P`.

The return Statement

The **return** statement ends the execution of the function in which it appears. It can also return a value to the calling function. For example:

```
return;        /* End function and return no value */

return myvariable; /* End function and return value of myvariable */
```

The switch Statement

The **switch** statement allows you to branch to various sections of code based on the value of a single variable. This variable must evaluate to a **char**, **int**, or **long** constant.

Each section of code in the **switch** statement is marked with a case label—the keyword **case** followed by a constant or constant expression. The value of the **switch** test expression is compared to the constant in each case label. If a match is found, control transfers to the statement after the matching label and continues until you reach a **break** statement or the end of the **switch** statement.

For example:

```
switch( answer )
{
   case 'y': /* First case */
      printf( "lowercase y\n" );
      break;
```

```
   case 'n': /* Another case */
      printf( "lowercase n\n" );
      break;

   default:  /* Default case */
      printf( "not a lowercase y or n\n" );
      break;
}
```

The example tests the value of the variable `answer`. If `answer` evaluates to the constant `'y'`, control transfers to the first case in the **switch** statement. If it equals `'n'`, control transfers to the second case.

A case labelled with the **default** keyword executes when none of the other case constants matches the value of the **switch** test expression. In the example, the **default** case executes when `answer` equals any value other than `'y'` or `'n'`.

If you omit the **break** statement at the end of a case, execution falls through to the next case.

If you omit the **default** case and no matching case is found, nothing in the **switch** statement executes.

No two **case** constants in the same **switch** statement can have the same value.

The while Statement

The **while** statement repeats a statement until its test expression becomes false. A **while** loop evaluates its test expression before executing its loop body. If the test expression is false when the loop begins, the loop body never executes. (Contrast this behavior with the **do** loop, which always executes its loop body at least once.)

For example:

```
while( !sample )  /* Repeat until sample equals 1 */
{
   printf( "%d\t%d\n", x, x*x );
   x += 6;
   if( x > 20 )
      sample = 1;
}
```

You can exit a **while** loop early with a **break** or **goto** statement. The **continue** statement skips to the next iteration of the loop.

Data Types

Data types are explained in Chapter 4, "Data Types," and Chapter 5, "Advanced Data Types." A brief description is given here.

Basic Data Types

The basic data types in C are character (**char**), integer (**int**), and floating point (**float** and **double**). All other data types are derived from these basic types. For example, a string is an array of **char** values.

Table A.1 lists the range of values for each data type.

Table A.1 Basic Data Types

Type Name	Other Names	Range of Values
char	**signed char**	–128 to 127
unsigned char	none	0 to 255
int	**signed, signed int**	–32,768 to 32,767
unsigned	**unsigned int**	0 to 65,535
unsigned short	**unsigned short int**	0 to 65,535
short	**short int, signed short**	–32,768 to
	signed short int	32,767
long	**long int, signed long**	–2,147,483,647 to
	signed long int	2,147,483,648
unsigned long	**unsigned long int**	0 to 4,294,967,295
_segment	none	0 to 65,535
enum	none	–32,768 to 32,767
float	none	Approximately 1.2E–38 to 3.4E+38 (7-digit precision)
double	none	Approximately 2.2E–308 to 1.8E+308 (15-digit precision)
long double	none	Approximately 3.4E–4932 to 1.2E+4932 (19-digit precision)

Character Type

The character type (**char**) occupies one byte of storage and can express a whole number in the range of –128 to 127. Unsigned characters have a range of 0 to 255. You can represent any ASCII character as an **unsigned char** value.

Typical declarations of character types are shown below:

```
char answer; /* Declare a character variable answer */

char alpha = 'a'; /* Declare character variable alpha
                     and initialize it */
```

A character constant represents a single ASCII character. Typical character constants are shown below:

```
char alpha = 'a'; /* Declare and initialize */

char c2 = 0x61; /* Declare and initialize with
                   hexadecimal value for 'a' */
```

Escape Sequences Escape sequences represent special characters, such as the carriage return. An escape sequence consists of a backslash character plus a letter or punctuation mark. Table A.2 lists the C escape sequences; they are also listed in online help.

Table A.2 C Escape Sequences

Character	Meaning	Hexadecimal Value
\a	Alert (bell)	0x07
\n	New line (linefeed)	0x0A
\b	Backspace	0x08
\r	Carriage return	0x0D
\f	Formfeed	0x0C
\t	Tab	0x09
\v	Vertical tab	0x0B
\\	Backslash	0x5C
\'	Single quote	0x27
\"	Double quote	0x22
\0	Null	0x00

Integer Type

The integer (**int**) type occupies two bytes of storage and can express a whole number in the range –32,768 to 32,767. Unsigned integers (**unsigned** or **unsigned int**) have a range of 0 to 65,535.

In QuickC, short integers (**short** or **short int**) are the same as integers (**int**). Note that the **short** and **int** types are not the same in some operating systems other than DOS.

Signed long integers (**long**) occupy four bytes and have a range of –2,147,483,648 to 2,147,483,647. Unsigned long integers have a range of 0 to 4,294,967,295.

Integer variables are declared with the keywords **int**, **short**, **unsigned**, or **long**. Typical declarations of integer types are shown below:

```
int z; /* Declare an int variable z */

int ten = 10; /* Declare int variable and
                 assign it the value 10 */

unsigned int a; /* Declare unsigned int variable */

unsigned long BigInt = 2000000001UL; /* Declare and
                                        initialize */
```

Integer constants are used to represent decimal, octal, and hexadecimal numbers. There are three types of integer constants:

1. Decimal constants can only contain the digits 0–9. The first digit must not be 0.
2. Octal constants can only contain the digits 0–7. The first digit must be 0.
3. Hexadecimal constants can only contain the digits 0–9, plus the letters a–f or A–F. The constant must begin with either 0x or 0X.

You can specify that an integer constant is long by adding the suffix **l** or **L**. The suffix can be used with decimal, hexadecimal, or octal notation.

To specify that an integer constant is short, add the suffix **u** or **U**. This suffix can also be used with decimal, hexadecimal, or octal notation.

Typical integer constants are shown below:

```
42    /* Decimal constant */

0x34  /* Hexadecimal constant */

0x3cL /* Long hexadecimal constant */
```

```
052   /* Octal constant */
```

Floating-Point Types

You can declare floating-point variables using the keywords **float** or **double**. The **float** type occupies four bytes of storage and can express a floating-point value in the range 1.2E–38 to 3.4E+38. This type has seven-digit precision.

The **double** type occupies eight bytes of storage and can express a floating-point value in the range 2.2E–308 to 1.8E+308. This type has fifteen-digit precision.

The **long double** type occupies ten bytes of storage and can express a floating-point value in the range 3.4E–4932 to 1.2E+4932. This type has nineteen-digit precision.

Typical declarations of floating-point types are shown below:

```
float SmallPi = 3.14;  /* Declare floating-point variable */

double AccuratePi = 3.141592653596  /* Declare
                                       double-precision */
```

Floating-point constants can represent decimal numbers in either single or double precision. A floating-point constant must either contain a decimal point or end with the suffix **e** or **E**. Typical floating-point constants are shown below:

```
2.78   /* Floating-point constant */

3E     /* Floating-point constant */
```

Aggregate Data Types

Aggregate data types are built from one or more of the basic data types. These include the following:

- Arrays (including strings)
- Structures
- Unions

Arrays and Strings

An "array" is a collection of data elements of a single type. An array can contain any data type. You can access an element of an array by using the array name and a numeric subscript.

A "string" is an array of characters that terminates with the null character (**\0**). Arrays that contain strings must allow space for the final null character.

Typical arrays and strings are shown below:

```
int id_number[10]; /* One-dimensional;
                      10 elements; integer */

char name[30]; /* String */

float matrix[5][3]; /* Two-dimensional array,
                       5 rows, 3 columns */

char baby[30] = "Peter Roddy"; /* String initialization */
```

Structures

A "structure" is a collection of data items of different types. Once you have defined a structure type, you can declare a structure variable using that type.

The following example illustrates a simple structure:

```
struct date
{
   int month;
   int day;
   int year;
}

struct date today;
```

The example defines a structure type named `date` and declares a structure variable `today` to be of type `date`.

Use the structure-member operator (.) to access the "elements" (members) of a structure. The name

```
today.month
```

refers to the `month` member of the `today` structure in the example.

Unions

A "union" is a set of data items of different types sharing the same storage space in memory. One use of unions is accessing the computer's DOS registers. For instance, QuickC defines the union **REGS** as the following:

```
union REGS
{
   struct WORDREGS x;
   struct BYTEREGS h;
};
```

Advanced Data Types

Advanced data topics are explained in Chapter 5, "Advanced Data Types." A brief description of each topic is given here.

Visibility

Variables declared outside all functions are global and can be accessed anywhere in the current source file. Variables declared inside a function are local and can be accessed only in that function. Use the **extern** keyword to make a variable declared in another source file visible in the current source file.

Lifetime

Global variables, and local variables declared with the **static** keyword, exist for the lifetime of the program. Other local variables are "automatic;" they come into being when the function starts and evaporate when it ends.

Type Conversions

A type conversion occurs automatically when an expression mixes two different data types. QuickC converts the lower-ranking type to the higher-ranking type before it performs the specified operation.

You can also "cast" (manually convert) a value to any type by placing the desired type name in parentheses in front of the value. The example below casts the value of `sample` to type **float** and assigns the value to `x`:

```
int sample;
float x;
x = (float)sample;
```

User-Defined Types

The **typedef** keyword allows you to create user-defined types, which are synonyms for existing data types. User-defined types can make your program more readable. For example, a type called `string` may be easier to understand than a type called **char ***.

A simple **typedef** declaration is shown below. The name of an existing type (**long int**) is followed by the synonym `income`.

```
typedef long int income;
```

Once you have created a new type name, you can use it wherever the original type name could be used:

```
income net_income, gross_income;
```

In the example above, the variables `net_income` and `gross_income` are of type `income`, which is the same as **long int**.

Enumerated Types

An enumerated type (declared with **enum**) has values limited to a specified set. If the **enum** declaration does not specify any values, QuickC assigns sequential integers to the enumeration identifiers beginning at zero.

The example below assigns the values of 0, 1, and 2 to the enumeration identifiers `zero`, `one`, and `two`, respectively. It also creates an enumerated type `small_numbers` that can be used to declare other variables.

```
/* Enumerated data type */
enum small_numbers {zero, one, two};

/* Variable my_numbers is of type small_numbers */
enum small_numbers  my_numbers;
```

The following example explicitly assigns values to the enumeration identifiers:

```
/* Enumerated data type */
enum even_numbers { two = 2, four = 4, six = 6 };
```

Operators

C-language operators are explained in Chapter 6, "Operators."

An "operand" is a constant or variable manipulated by an operator in an expression. An "operator" specifies how the operands in an expression are to be evaluated. Operators also produce a result that can be nested within a larger expression.

C provides a rich set of operators covering everything from basic arithmetic operations to logical and bitwise operations. You can also combine the assignment operator (=) with any arithmetic or bitwise operator to form a combined assignment operator.

C operators have two properties, precedence and associativity. You can change the normal order of evaluation by enclosing expressions in parentheses.

Table A.3 lists the C operators and their precedence and associativity values. The lines in the table separate precedence levels. The highest precedence level is at the top of the table.

Table A.3 C Operators

Symbol	Name or Meaning	Associativity
( )	Function call	Left to right
[ ]	Array element	
.	Structure or union member	
->	Pointer to structure member	
--	Decrement	Right to left
++	Increment	
:>	Base operator	Left to right
!	Logical NOT	Right to left
~	One's complement	
-	Unary minus	
+	Unary plus	
&	Address	
*	Indirection	
sizeof	Size in bytes	
(type)	Type cast [for example, (**float**) i]	
*	Multiply	Left to right
/	Divide	
%	Modulus (remainder)	
+	Add	Left to right
-	Subtract	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal to	
>	Greater than	
>=	Greater than or equal to	
==	Equal	Left to right
!=	Not equal	

Table A.3 C Operators (*continued*)

<table>
<tr><th>Symbol</th><th>Name or Meaning</th><th>Associativity</th></tr>
<tr><td>&</td><td>Bitwise AND</td><td>Left to right</td></tr>
<tr><td>^</td><td>Bitwise exclusive OR</td><td>Left to right</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>Left to right</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>Left to right</td></tr>
<tr><td>||</td><td>Logical OR</td><td>Left to right</td></tr>
<tr><td>? :</td><td>Conditional</td><td>Right to left</td></tr>
<tr><td>=
*=, /=, %=, +=, –=
<<=, >>=, &=, ^=, |=</td><td>Assignment
Compound assignment</td><td>Right to left</td></tr>
<tr><td>,</td><td>Comma</td><td>Left to right</td></tr>
</table>

Preprocessor Directives

Preprocessor directives are explained in Chapter 7, "Preprocessor Directives."

A "preprocessor directive" is a command to the QuickC compiler, which processes all such commands before it compiles your source program. A preprocessor directive begins with the # symbol, followed by the directive and any arguments the directive needs. Since a preprocessor directive is not a C language statement, it doesn't end in a semicolon.

The two most commonly used directives are **#define** and **#include**. Use the **#define** directive to give a meaningful name to some constant in your program. The following directive tells QuickC to replace `PI` with `3.14159` everywhere in the source program:

```
#define PI  3.14159
```

The **#include** directive below tells QuickC to insert the contents of a specified file at the current location in your source program.

```
#include <stdio.h>      /* Standard header file */
```

Such files are called "include files" or "header files." Standard header files, such as STDIO.H, end with the .H extension and contain function prototypes and other definitions needed for QuickC library routines.

Table A.4 lists and describes the QuickC standard header files. Consult online help for information on the header files needed by individual library functions.

Table A.4 QuickC Header Files

File Name	Major Contents
ASSERT.H	**assert** debugging macro
BIOS.H	BIOS service functions
CONIO.H	Console and port I/O routines
CTYPE.H	Character classification
DIRECT.H	Directory control
DOS.H	MS-DOS interface functions
ERRNO.H	System-wide error numbers
FCNTL.H	Flags used in **open** and **sopen** functions
FLOAT.H	Constants needed by math functions
GRAPH.H	Low-level graphics and font routines
IO.H	File-handling and low-level I/O
LIMITS.H	Ranges of integers and character types
LOCALE.H	Internationalization functions
MALLOC.H	Memory-allocation functions
MATH.H	Floating-point math routines
MEMORY.H	Buffer-manipulation routines
PGCHART.H	Presentation graphics
PROCESS.H	Process-control routines
SEARCH.H	Searching and sorting functions
SETJMP.H	**setjmp** and **longjmp** functions
SHARE.H	Flags used in **sopen**
SIGNAL.H	Constants used by signal function
STDARG.H	Macros used to access variable-length argument-list functions
STDDEF.H	Commonly used data types and values
STDIO.H	Standard I/O header file
STDLIB.H	Commonly used library functions
STRING.H	String-manipulation functions
TIME.H	General time functions
VARARGS.H	Variable-length argument-list functions
SYS\LOCKING.H	Flags used by **locking** function

Table A.4 QuickC Header Files (*continued*)

File Name	Major Contents
SYS\STAT.H	File-status structures and functions
SYS\TIMEB.H	**time** function
SYS\TYPES.H	File-status and time types
SYS\UTIME.H	**utime** function

Pointers

Pointers are described in Chapter 8, "Pointers," and Chapter 9, "Advanced Pointers."

A "pointer" is a variable that contains the memory address of an item rather than its value. A pointer can point to any type of data item or to a function. The following code illustrates pointer declarations:

```
int *intptr; /* Pointer to an integer */
char *name; /* Pointer to char */
```

The following operators are used with pointers:

- The indirection operator (*) has two uses. In a declaration, it means that the declared item is a pointer. In an expression, it denotes the data being pointed to.
- The address-of operator (&) yields the memory address at which an item is stored.

You can perform four arithmetic operations on pointers:

1. Adding a pointer and an integer
2. Subtracting an integer from a pointer
3. Subtracting two pointers
4. Comparing two pointers

Pointer arithmetic operations are automatically scaled by the size of the object pointed to. For instance, adding 1 to a pointer to a **float** item causes the address stored in the pointer to be incremented four bytes, the size of one **float** item.

QuickC 2.5 also supports based pointers, a highly advanced feature, that are compatible with Microsoft C version 6.0. Please refer to your C 6.0 documentation for more information about based pointers and objects.

Appendix B
C Library Guide

This appendix outlines the features of the C run-time library provided with QuickC. It does not intend to be a complete presentation of the complete C run-time library. Instead, this appendix presents the most fundamental routines, grouped by category so you can begin experimenting with C and with QuickC.

Remember, use online help to get instant help on any topic of interest. The online help system provided with QuickC provides complete reference information for all C library functions, keywords, and preprocessor directives.

Overview of the C Run-Time Library

At last count, the C run-time library contained over 400 functions to use in C programs. This appendix describes the major categories of functions included in the library and, within those categories, the fundamental routines every C programmer should know.

The discussions of these categories give only a brief overview of the capabilities of the run-time library. You can find a complete description of the syntax and use of each routine in online help.

The routines in the C run-time library are divided into the following categories:

Table B.1 C Run-Time Library Routines

Category	Function Routines	Page
Buffer Manipulation	memchr, memcmp, memcpy, memmove, memset	345
Character Classification and Conversion	isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper	346
Data Conversion	atof, atoi, atol, itoa, ltoa, ultoa, strtod, strtol, strtoul	348
Error Message	assert, perror, strerror, _strerror	349

Table B.1 C Run-Time Library Routines (*continued*)

Category	Function Routines	Page
Graphics 1: Low-Level Graphics		350
Configure Mode and Environment	_displaycursor, _getvideoconfig, _setvideomode	351
Set Coordinates	_getcurrentposition, _getphyscoord,_getviewcoord, _getwindowcoord, _setcliprgn,_setvieworg, _setviewport, _setwindow	352
Set Palette	_remapallpalette, remappallette, _selectpalette	354
Set Attributes	_getbkcolor, _getcolor, _setbkcolor, _setcolor	355
Output Images	_arc, _clearscreen, _ellipse, _floodfill, _getpixel, _lineto, _moveto, _pie, _rectangle, _setpixel	356
Output Text	_displaycursor, _gettextcolor, _gettextcursor, _gettextposition, _outtext, _settextposition, _settextcolor, _settextwindow	359
Transfer Images	_getimage, _imagesize, _putimage	361
Graphics 2: Presentation Graphics	_pg_chart, _pg_chartms, _pg_chartpie, _pg_chartscatter, _pg_chartscatterms, _pg_defaultchart, _pg_initchart	362
Graphics 3: Font Display	_getfontinfo, _getgtextextent, _outgtext, _registerfonts, _setfont, _unregisterfonts	365
Input and Output		367
Stream Routines	clearerr, fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell, fwrite, getc, getchar, gets, printf, putc, putchar, puts, rewind, scanf, sprintf, sscanf, tmpfile, tmpnam, ungetc	367
Low-Level Routines	close, creat, eof, lseek, open, read, tell, write	373
Console and Port I/O Routines	cgets, cprintf, cputs, cscanf, getch, getche, kbhit, putch, ungetch	375
Math	abs, fabs, labs, acos, asin, atan, atan2, ceil, cos, cosh, exp, floor, fmod, frexp, ldexp, log, log10, modf, pow, rand, srand, sin, sinh, sqrt, tan, tanh	377
Memory Allocation	calloc, free, _ffree, hfree, _nfree, malloc, _fmalloc, _nmalloc, realloc	381
Process Control	abort, atexit, exit, _exit, system	383

Table B.1 C Run-Time Library Routines *(continued)*

Category	Function Routines	Page
Searching and Sorting	bsearch, lfind, lsearch, qsort	384
String Manipulation	strcat, strcpy, strdup, strncat, strncpy, strchr, strcspn, strpbrk, strrchr, strspn, strstr, strcmp, strcmpi, stricmp, strncmp, strnicmp, strlen, strlwr, strupr, strnset, strset, strtok	385
Time	asctime, clock, ctime, difftime, ftime, gmtime, mktime, time	389

Buffer-Manipulation Routines

Buffer manipulation routines are used with areas of memory on a character-by-character basis. Buffers are arrays of characters (bytes). Unlike strings, however, they are not usually terminated with a null character (**\0**).

memchr

Returns a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer.

Include STRING.H

Prototype **void *memchr(const void ****buf***, int** *c***, size_t** *count* **);**

Arguments

buf	Pointer to buffer
c	Character to copy
count	Number of characters

Returns A pointer to the location of *c* in *buf* if successful; NULL if *c* is not within first *count* bytes of *buf*

memcmp

Compares a specified number of characters from two buffers.

Include STRING.H

Prototype **int memcmp(const void ****buf1***, const void ****buf2***, size_t** *count* **);**

Arguments

buf1	First buffer
buf2	Second buffer
count	Number of characters

Returns A negative value if *buf1* < *buf2*, 0 if *buf1* = *buf2*, a positive value if *buf1* > *buf2*

memcpy Copies a specified number of characters from one buffer to another.

Include	STRING.H	
Prototype	**void *memcpy(void ****dest***, const void ****src***, size_t** *count* **);**	
Arguments	*dest*	New buffer
	src	Buffer to copy from
	count	Number of characters to copy
Returns	A pointer to *dest*	

memmove Copies a specified number of characters from one buffer to another. When the source and target areas overlap, the **memmove** function is guaranteed to properly copy the full source.

Include	STRING.H	
Prototype	**void *memmove(void ****dest***, const void ****src***, size_t** *count* **);**	
Arguments	*dest*	Target object
	src	Source object
	count	Number of characters to copy
Returns	The value of *dest*	

memset Uses a given character to initialize a specified number of bytes in the buffer.

Include	STRING.H	
Prototype	**void *memset(void ****dest***, int** *c***, size_t** *count* **);**	
Arguments	*dest*	Pointer to destination
	c	Character to set
	count	Number of characters
Returns	A pointer to *dest*	

Character Classification and Conversion Routines

The classification routines (**is...**) test a character and return a one (1) if the character is in the set that the routine is testing for. The conversion routines (**to...**) convert characters between uppercase and lowercase. These routines are generally faster than writing a test expression such as the following:

```
if ((c >= 0) || c <= 0x7f))
```

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit These routines test a character for a specified condition and return a nonzero value if the condition is true.

Include	CTYPE.H
Prototypes	**int isalnum(int** *c* **);** (alphanumeric character: A–Z, a–z, 0–9)
	int isalpha(int *c* **);** (alphabetic character: A–Z, a–z)
	int isascii(int *c* **);** (ASCII character: 0x00–0x7F)
	int iscntrl(int *c* **);** (control character: 0x00–0x1F, 0x7F)
	int isdigit(int *c* **);** (decimal digit: 0–9)
	int isgraph(int *c* **);** (printable character, not space: 0x21–0x7E)
	int islower(int *c* **);** (lowercase letter: a–z)
	int isprint(int *c* **);** (printable character: 0x20–0x7E)
	int ispunct(int *c* **);** (punctuation character)
	int isspace(int *c* **);** (white-space character: 0x09–0x0D, 0x20)
	int isupper(int *c* **);** (uppercase letter: A–Z)
	int isxdigit(int *c* **);** (hexadecimal digit: A–F, a–f, 0–9)
Argument	*c* Character to be tested
Returns	A nonzero value if the condition is true

tolower, toupper These routines accept a character argument and return a converted character. The **tolower** and **toupper** routines are also implemented as functions. To use the function versions, you must do the following:

- Include CTYPE.H if necessary for other macro definitions
- If CTYPE.H is included, give **#undef** directives for **tolower** and **toupper**
- Include STDLIB.H (which contains the function prototypes)

Include	CTYPE.H
Prototypes	**int tolower(int** *c* **);**
	int toupper(int *c* **);**
Argument	*c* Character to be converted
Returns	**tolower**: the lowercase equivalent of *c* only if *c* is an uppercase letter **toupper**: the uppercase equivalent of *c* only if *c* is a lowercase letter

Data Conversion Routines

The data conversion routines convert numbers to strings of ASCII characters and vice versa.

atof, atoi, atol

These ASCII-to-number routines convert an ASCII string to a **float**, an **integer**, and a **long**, respectively.

Include	STDLIB.H or MATH.H (**atof**) STDLIB.H (**atoi, atol**)
Prototypes	**double atof(const char** **string* **);** **int atoi(const char** **string* **);** **long atol(const char** **string* **);**
Argument	*string* String to be converted
Returns	The converted string, or 0 if *string* cannot be converted

itoa, ltoa, ultoa

These number-to-ASCII routines convert an integer, a long value, or an unsigned long value to an ASCII string.

Include	STDLIB.H
Prototypes	**char * itoa(int** *value*, **char** **string*, **int** *radix* **);** **char * ltoa(long** *value*, **char** **string*, **int** *radix* **);** **char * ultoa(unsigned long** *value*, **char** **string*, **int** *radix* **);**
Arguments	*value* Number to be converted *string* String result *radix* Number base of value
Returns	A pointer to *string*; there is no error return

strtod, strtol, strtoul

These routines convert a string to a **double**, a **long**, and an **unsigned long**, respectively.

Include	STDLIB.H
Prototypes	**double strtod(const char** **nptr*, **char** ***endptr* **);** **long strtol(const char** **nptr*, **char** ***endptr*, **int** *base* **);** **unsigned long strtoul(const char** **nptr*, **char** ***endptr*, **int** *base* **);**
Arguments	*nptr* String to convert *endptr* End of scan *base* Number base to use

Returns	**strtod**: the converted value; overflow returns **HUGE_VAL**, underflow returns 0 **strtol**: the converted value; overflow returns **LONG_MAX** or **LONG_MIN**, depending on sign of converted value **strtoul**: the converted value if successful, 0 if not, and **ULONG_MAX** on overflow

Error Message Routines

The routines in this category handle the display of error messages.

The **assert** macro is typically used to test for program logic errors; it prints a message when a given "assertion" fails to hold true. Defining the identifier **NDEBUG** to any value causes occurrences of **assert** to be disabled in the source file, thus allowing you to turn off assertion checking without modifying the source file.

The **perror** routine prints the system-error message, along with a user-supplied message, for the last system-level call that produced an error. The **perror** routine is declared in the include files STDLIB.H and STDIO.H. The error number is obtained from the **errno** variable. The system message is taken from the **sys_errlist** array.

The **strerror** and **_strerror** routines store error messages in a string.

assert Tests for logic error.

Include	ASSERT.H, STDIO.H	
Prototype	**void assert(** *expression* **);**	
Argument	*expression*	Expression to test
Returns	Void	

perror Prints error message.

Include	STDIO.H	
Prototypes	**void perror(const char** **string* **);** **int** *errno*; **int** *sys_nerr*; **char** **sys_errlist* [*sys_nerr*];	
Arguments	*string*	User-supplied message
	errno	Error number
	sys_nerr	Number of system-error messages
	sys_errlist	Array of error messages
Returns	Void	

strerror, _strerror Saves system-error message and optional user-error message in string. The routine **strerror** is the ANSI-compatible version.

Include	STRING.H	
Prototypes	**char *strerror(int** *errnum* **);** **char *_strerror(char** **string* **);** **int** *errno***;** **int** *sys_nerr***;** **char** **sys_errlist* [*sys_nerr*]**;**	
Arguments	*errnum*	Error number
	string	User-supplied message
	errno	Error number
	sys_nerr	Number of system-error messages
	sys_errlist	Array of error messages
Returns	A pointer to the error-message string	

Graphics 1: Low-Level Graphics Routines

The low-level graphics routines provide line, figure, and pixel manipulation capabilities. The routines for presentation graphics are described in the next section. The routines for displaying fonts follow the presentation graphics section.

The graphics package supports the IBM® (and compatible) Enhanced Graphics Adapter (EGA), Color Graphics Adapter (CGA), certain operating modes of the Video Graphics Array (VGA) hardware configurations, and the MCGA (Multicolor Graphics Array). The graphics package also supports the Hercules Graphics Card, Graphics Card Plus, InColor Card, and 100 percent compatible cards, as well as the special Olivetti® modes available on AT&T® computers.

The low-level graphics routines can be divided into the seven categories listed below, corresponding to the different tasks involved with creating and manipulating graphic objects:

Category	Task
Configure mode and environment	Selects the proper display mode for the hardware and establishes memory areas for writing and displaying images
Set coordinates	Specifies the logical origin and the active display area within the screen
Set palette	Specifies a palette mapping
Set attributes	Specifies background and foreground colors and mask and line styles

Output images	Draws and fills figures on the screen
Output text	Writes text to the screen
Transfer images	Stores images in memory and retrieves them

Configure Mode and Environment

The configure category of functions sets the status of the cursor, sets active and visual pages, and determines and sets video display modes.

The **_setvideomode** and **_getvideoconfig** functions are generally used at the very beginning of a graphics program.

_displaycursor

Determines whether the cursor will be left on or turned off on exit from graphics routines.

Include	GRAPH.H
Prototype	**short _far _displaycursor(short** *toggle* **);**
Argument	*toggle* Cursor state (**_GCURSORON**, **_GCURSOROFF**)
Returns	The previous value of toggle

_getvideoconfig

Obtains status of current graphics environment.

Include	GRAPH.H
Prototype	**struct videoconfig _far * _far _getvideoconfig** **(struct videoconfig _far** **config* **);**
Argument	*config* Configuration information
Returns	The configuration information as a **videoconfig** structure

_setvideomode

Selects screen display mode.

Include	GRAPH.H
Prototype	**short _far _setvideomode(short** *mode* **);**
Argument	*mode* Desired mode (**_DEFAULTMODE, _TEXTBW40, _TEXTC40, _TEXTBW80, _TEXTC80, _MRES4COLOR, _MRESNOCOLOR, _HRESBW, _TEXTMONO, _HERCMONO, _MRES16COLOR, _HRES16COLOR, _ERESNOCOLOR, _ERESCOLOR, _VRES2COLOR, _VRES16COLOR, _MRES256COLOR, _ORESCOLOR, _MAXRESMODE, _MAXCOLORMODE**)
Returns	A nonzero value if successful, 0 if not

Set Coordinates

The Microsoft C graphics routines recognize three sets of coordinates:

1. Window coordinates defined with real-number values that are mapped to a specified viewport
2. Viewport coordinates defined by the application (viewport coordinates)
3. Fixed physical coordinates determined by the hardware and display configuration of the user's environment (physical coordinates)

The functions in this category alter the coordinate systems and provide a means to translate coordinates between the various systems.

Most of these routines have two or three forms. The functions are listed by the "base" name, without a suffix. Note, though, that function names that end with a **_w**, such as **_getcurrentposition_w**, use the window-coordinate system. Those that end with a **_wxy**, such as **_getviewcoord_wxy**, use the window-coordinate system and a **_wxycoord** structure to define the coordinates.

The default viewport-coordinate system is identical to the physical one. The physical origin (0, 0) is always in the upper left corner of the display. The *x* axis extends in the positive direction left to right, and the *y* axis extends in the positive direction top to bottom.

The dimensions of the *x* and *y* axes depend upon the hardware display configuration and the selected mode. These values are accessible at run time by examining the **numxpixels** and **numypixels** fields of the **videoconfig** structure returned by **_getvideoconfig**.

_getcurrentposition

Obtains the coordinates of the current graphic-output position. The **_getcurrentposition** function returns the position as an **xycoord** structure and the **_getcurrentposition_w** function returns the position as a **_wxycoord** structure.

Include	GRAPH.H
Prototypes	**struct xycoord _far _getcurrentposition(void);** **struct _wxycoord _far _getcurrentposition_w(void);**
Arguments	None
Returns	**_getcurrentposition**: the coordinates of the current position as an **xycoord** structure **_getcurrentposition_w**: the coordinates of the current position as a **_wxycoord** structure

_getphyscoord Converts viewport coordinates to physical coordinates.

Include GRAPH.H

Prototype **struct xycoord _far _getphyscoord(short** *x*, **short** *y* **);**

Argument *x, y* View point to translate

Returns A pair of physical coordinates as an **xycoord** structure

_getviewcoord Converts specified coordinates to viewport coordinates.

Include GRAPH.H

Prototypes **struct xycoord _far _getviewcoord(short** *x*, **short);**

struct xycoord _far _getviewcoord_w(double *wx*, **double** *wy* **);**

struct xycoord _far _getviewcoord_wxy(struct _wxycoord _far **pwxyl* **);**

Arguments
x, y Physical point to translate
wx, wy Window-coordinate point to translate
pwxyl Window-coordinate point to translate

Returns A pair of logical coordinates as an **xycoord** structure

_getwindowcoord Converts physical coordinates to window coordinates.

Include GRAPH.H

Prototype **struct _wxycoord _far _getwindowcoord(short** *x*, **short** *y* **);**

Argument *x, y* Physical point to translate

Returns A pair of window coordinates as a **_wxycoord** structure

_setcliprgn Limits graphic output to part of the screen.

Include GRAPH.H

Prototype **void _far _setcliprgn(short** *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

Arguments
x1, y1 Upper left corner of clip region
x2, y2 Lower right corner of clip region

Returns Void

_setvieworg Positions the logical origin.

Include GRAPH.H

Prototype **struct xycoord _far _setvieworg(short** *x*, **short** *y* **);**

Argument *x, y* New origin point

Returns The physical coordinates of the previous viewport origin in an **xycoord** structure

_setviewport

Limits graphic output and positions the logical origin within a limited area.

Include GRAPH.H

Prototype **void _far _setviewport(short** *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

Arguments

x1, y1	Upper left corner of window
x2, y2	Lower right corner of window

Returns Void

_setwindow

Defines a window-coordinate system.

Include GRAPH.H

Prototype **void _far _setwindow(short** *finvert*, **double** *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2* **);**

Arguments

wx1, wy1	Upper left corner of window
wx2, wy2	Lower right corner of window
finvert	Invert flag (TRUE, FALSE)

Returns Void

Set Palette

A screen pixel can be represented as a one-, two-, or four-bit value, depending on the particular mode. The byte representation is called the "color value."

Each color that can be displayed is represented by a unique ordinal value called a "color index." A palette is simply a mapping of the actual display colors to the legal values.

_remapallpalette, _remappalette

The **_remapallpalette** routine assigns colors to all color values. The **_remappalette** routine assigns color indexes to selected color values.

Include GRAPH.H

Prototypes **short _far _remapallpalette(long _far** **colors* **);**

long _far _remappalette(short *index*, **long** *color* **);**

Arguments

colors	Color value array: (**_BLACK, _BLUE, _GREEN, _CYAN, _RED, _MAGENTA, _BROWN, _WHITE, _GRAY, _LIGHTBLUE,, _LIGHTGREEN, _LIGHTCYAN, _LIGHTRED, _LIGHTMAGENTA, _LIGHTYELLOW, _BRIGHTWHITE**)
index	Color index to reassign
color	Color value to assign color index

Returns **_remapallpalette:** 0 if successful, –1 if not
_remappalette: the previous color value of the index argument if successful, –1 if not

_selectpalette

Selects a predefined palette.

Include	GRAPH.H
Prototype	**short _far _selectpalette(short** *number* **);**
Argument	*number* Palette number
Returns	The value of the previous palette

Set Attributes

Attributes are characteristics (color, fill pattern, or line style) that can be specified for low-level graphics routines.

A fill mask is an 8-by-8-bit template array, with each bit representing a pixel. If a bit is 0, the pixel in memory is left untouched: the mask is transparent to that pixel. If a bit is 1, the pixel is assigned the current color value. The template is repeated over the entire fill area.

A line style is a 16-bit template buffer, with each bit corresponding to a pixel. If a bit is 0, the pixel is set to the current background color. If a bit is 1, the pixel is set to the current color. The template is repeated for the length of the line.

_getbkcolor

Reports the current background color.

Include	GRAPH.H
Prototype	**long _far _getbkcolor(void);**
Arguments	None
Returns	The current background color

_getcolor

Obtains the current color.

Include	GRAPH.H
Prototype	**short _far _getcolor(void);**
Arguments	None
Returns	The current color

_setbkcolor

Sets the current background color.

Include	GRAPH.H
Prototype	**long _far _setbkcolor(long** *color* **);**
Argument	*color* Desired color value
Returns	The color value of the previous background color

_setcolor Sets the current color.

Include	GRAPH.H
Prototype	**short _far _setcolor(short** *color* **);**
Argument	*color* Desired color index
Returns	The previous color

Output Images

These routines display graphic elements (arcs, lines, pixels, etc.) on the screen.

Circular figures such as arcs and ellipses are centered within a "bounding rectangle," specified by two points that define the diagonally opposed corners of the rectangle. The center of the rectangle becomes the center of the figure, and the rectangle's borders determine the size of the figure.

Most of these routines have two or three forms. The functions are listed by the "base" name, without a suffix. Note, though, that function names that end with a **_w**, such as **_arc_w**, use the window coordinate system. Those that end with a **_wxy**, such as **_ellipse_wxy**, use the window coordinate system and a **_wxycoord** structure to define the coordinates.

_arc Draws an arc.

Include GRAPH.H

Prototypes

short _far _arc(short *x1*, **short** *y1*, **short** *x2*, **short** *y2*,
short *x3*, **short** *y3*, **short** *x4*, **short** *y4* **);**

short _far _arc_wxy(struct _wxycoord *pwxy1*,
struct _wxycoord* *pwxy2*, **struct _wxycoord*** *pwxy3*,
struct _wxycoord* *pwxy4* **);**

Arguments

x1, *y1*	Upper left corner of bounding rectangle
x2, *y2*	Lower right corner of bounding rectangle
x3, *y3*	Start vector
x4, *y4*	End vector
pwxy1	Upper left corner of bounding rectangle
pwxy2	Lower right corner of bounding rectangle
pwxy3	Start vector
pwxy4	End vector

Returns A nonzero value if the arc is drawn successfully, 0 if not

_clearscreen Erases the screen and fills it with the current background color.

Include	GRAPH.H
Prototype	**void _far _clearscreen(short** *area* **);**
Argument	*area* Target area (_GCLEARSCREEN, _GVIEWPORT, _GWINDOW)
Returns	Void

_ellipse

Draws an ellipse.

Include GRAPH.H

Prototypes

short _far _ellipse(short *control*, **short** *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

short _far _ellipse_w(short *control*, **double** *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2* **);**

short _far _ellipse_wxy(short *control*, **struct _wxycoord*** *pwxy1*, **struct _wxycoord*** *pwxy2* **);**

Arguments

control	Fill flag (_GFILLINTERIOR, _GBORDER)
x1, *y1*	Upper left corner of bounding rectangle (view coordinates)
x2, *y2*	Lower right corner of bounding rectangle (view coordinates)
wx1, *wy1*	Upper left corner of bounding rectangle (window coordinates)
wx2, *wy2*	Lower right corner of bounding rectangle (window coordinates)
pwxy1	Upper left corner of bounding rectangle (window coordinates)
pwxy2	Lower right corner of bounding rectangle (window coordinates)

Returns A nonzero value if the ellipse is drawn successfully, 0 if not

_floodfill

Fills an area of the screen with the current color.

Include GRAPH.H

Prototypes

short _far _floodfill(short *x*, **short** *y*, **short** *boundary* **);**

short _far _floodfill_w(double *wx*, **double** *wy*, **short** *boundary* **);**

Arguments

x, *y*	Start point (view coordinates)
wx, *wy*	Start point (window coordinates)
boundary	Boundary color

Returns A nonzero value if successful, 0 if not

_getpixel

Obtains a pixel's color index. The coordinates can be specified in either view coordinates (**_getpixel**) or in window coordinates (**_getpixel_w**).

Include GRAPH.H

Prototypes

short _far _getpixel(short *x*, **short** *y* **);**

short _far _getpixel_w(double *wx*, **double** *wy* **);**

Arguments

x, *y*	Pixel position (view coordinates)
wx, *wy*	Pixel position (window coordinates)

Returns The color index if successful, –1 if not

_lineto Draws a line from the current graphic output position to a specified point. The coordinate of the end point can be specified in either view coordinates (**_lineto**) or in window coordinates (**_lineto_w**).

Include	GRAPH.H
Prototypes	**short _far _lineto(short** *x*, **short** *y* **);** **short _far _lineto_w(double** *wx*, **double** *wy* **);**
Arguments	*x, y* End point (view coordinates) *wx, wy* End point (window coordinates)
Returns	A nonzero value if the line is drawn successfully, 0 if not

_moveto Moves the current graphic-output position to a specified point. The coordinates can be specified in either view coordinates (**_moveto**) or in window coordinates (**_moveto_w**).

Include	GRAPH.H
Prototypes	**struct xycoord _far _moveto(short** *x*, **short);** **struct _wxycoord _far _moveto_w(double** *wx*, **double** *wy* **);**
Arguments	*x, y* Target position (view coordinates) *wx, wy* Target position (window coordinates)
Returns	The logical coordinates of the previous position as an **xycoord** structure (**_moveto**) or as a **_wxycoord** structure (**_moveto_w**)

_pie Draws a figure shaped like a pie wedge.

Include	GRAPH.H
Prototypes	**short _far _pie(short** *control*, **short** *x1*, **short** *y1*, **short** *x2*, **short** *y2*, **short** *x3*, **short** *y3*, **short** *x4*, **short** *y4* **);** **short _far _pie_wxy(short** *control*, **struct _wxycoord*** *pwxy1*, **struct _wxycoord****pwxy2*, **struct _wxycoord*** *pwxy3*, **struct _wxycoord****pwxy4* **);**
Arguments	*control* Fill flag (**_GFILLINTERIOR, _GBORDER**) *x1, y1* Upper left corner of bounding rectangle (view coordinates) *x2, y2* Lower right corner of bounding rectangle (view coordinates) *x3, y3* Start vector(view coordinates) *x4, y4* End vector (view coordinates) *pwxy1* Upper left corner of bounding rectangle (window coordinates) *pwxy2* Lower right corner of bounding rectangle (window coordinates) *pwxy3* Start vector (window coordinates) *pwxy4* End vector (window coordinates)
Returns	A nonzero value if the pie is drawn successfully, 0 if not

_rectangle

Draws a rectangle.

Include GRAPH.H

Prototypes

short _far _rectangle(short *control*, **short** *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

short _far _rectangle_w(short *control*, **double** *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2* **);**

short _far _rectangle_wxy(short *control*, **struct _wxycoord****pwxy1*, **struct _wxycoord****pwxy2* **);**

Arguments

control	Fill flag (**_GFILLINTERIOR**, **_GBORDER**)
x1, *y1*	Upper left corner (view coordinates)
x2, *y2*	Lower right corner (view coordinates)
wx1, *wy1*	Upper left corner (window coordinates)
wx2, *wy2*	Lower right corner (window coordinates)
pwxy1	Upper left corner (window coordinates)
pwxy2	Lower right corner (window coordinates)

Returns A nonzero value if the rectangle is drawn successfully, 0 if not

_setpixel

Sets a pixel's color index.

Include GRAPH.H

Prototypes

short _far _setpixel(short *x*, **short** *y* **);**

short _far _setpixel_w(double *wx*, **double** *wy* **);**

Arguments

x, *y*	Target pixel (view coordinates)
wx, *wy*	Target pixel (window coordinates)

Returns The pixel's previous value if successful, –1 if not

Output Text

These routines provide text output in both graphics and text modes.

These functions recognize text window boundaries and should be used in applications using text windows.

No formatting capability is provided. If you want to output integer or floating-point values, you must convert the values into a string variable before calling these routines. All screen positions are specified as character-row and character-column coordinates.

_displaycursor Sets the cursor "on" or "off" on exit from a graphics routine.

Include	GRAPH.H
Prototype	**short _far _displaycursor(short** *toggle* **);**
Argument	*toggle* Cursor state (_GCURSORON, _GCURSOROFF)
Returns	The previous value of *toggle*

_gettextcolor Obtains the current text color.

Include	GRAPH.H
Prototype	**short _far _gettextcolor(void);**
Arguments	None
Returns	The color index of the current text color

_gettextcursor Obtains the current cursor attribute.

Include	GRAPH.H
Prototype	**short _far _gettextcursor(void);**
Arguments	None
Returns	The current cursor attribute

_gettextposition Obtains the current text-output position.

Include	GRAPH.H
Prototype	**struct rccoord _far _gettextposition(void);**
Arguments	None
Returns	The current text position as an **rccoord** structure

_outtext Outputs text to the screen at the current position.

Include	GRAPH.H
Prototype	**void _far _outtext(unsigned char _far** **text* **);**
Argument	*text* Text to be output
Returns	Void

_settextposition Relocates the current text position.

Include	GRAPH.H
Prototype	**struct rccoord _far _settextposition(short** *row*, **short** *col* **);**
Arguments	*row* Row coordinate of new output position *col* Column coordinate of new output position
Returns	The previous text position in an **rccoord** structure

_settextcolor

Sets the current text color.

Include GRAPH.H

Prototype **short _far _settextcolor(short** *index* **);**

Argument *index* Desired color index

Returns The value of the previous color

_settextwindow

Sets the current text-display window.

Include GRAPH.H

Prototype **void _far _settextwindow(short** *r1*, **short** *c1*, **short** *r2*, **short** *c2* **);**

Arguments
r1, *c1* Upper left corner of window
r2, *c2* Lower right corner of window

Returns Void

Transfer Images

These functions transfer screen images between memory and the display, using a buffer allocated by the application. You can use these functions to animate graphics elements on the screen.

Most of these routines have two or three forms. The functions are listed by the "base" name, without a suffix. Note, though, that function names that end in a _w, such as **_getimage_w**, use the window-coordinate system. Those that end with a _wxy, such as **_imagesize_wxy**, use the window-coordinate system and a **_wxycoord** structure to define the coordinates.

The **_imagesize** function is used to find the size in bytes of the buffer needed to store a given image.

_getimage

Stores a screen image in memory.

Include GRAPH.H

Prototypes

void _far _getimage(short *x1*, **short** *y1*,
short *x2*, **short** *y2*, **char _huge** **image* **);**

void _far _getimage_w(double *wx1*, **double** *wy1*,
double *wx2*, **double** *wy2*, **char _huge** **image* **);**

void _far _getimage_wxy(struct _wxycoord*pwxy1*,
struct _wxycoord*pwxy2*, **char _huge** **image* **);**

Arguments

x1, y1	Upper left corner of bounding rectangle (view coordinates)
x2, y2	Lower right corner of bounding rectangle (view coordinates)
wx1, wy1	Upper left corner of bounding rectangle (window coordinates)
wx2, wy2	Lower right corner of bounding rectangle (window coordinates)
pwxy1	Upper left corner of bounding rectangle (window coordinates)
pwxy2	Lower right corner of bounding rectangle (window coordinates)
image	Storage buffer for screen image

Returns Void

_imagesize

Returns image size in bytes.

Include GRAPH.H

Prototypes

long _far _imagesize(short *x1*, **short** *y1*, **short** *x2*, **short** *y2* **);**

long _far _imagesize_w(double *wx1*, **double** *wy1*, **double** *wx2*, **double** *wy2* **);**

long _far _imagesize_wxy(struct _wxycoord* *pwxy1*, **struct _wxycoord*** *pwxy2* **);**

Arguments

x1, y1	Upper left corner of bounding rectangle (view coordinates)
x2, y2	Lower right corner of bounding rectangle (view coordinates)
wx1, wy1	Upper left corner of bounding rectangle (window coordinates)
wx2, wy2	Lower right corner of bounding rectangle (window coordinates)
pwxy1	Upper left corner of bounding rectangle (window coordinates)
pwxy2	Lower right corner of bounding rectangle (window coordinates)

Returns The storage size of the image in bytes

_putimage

Retrieves an image from memory and displays it.

Include GRAPH.H

Prototypes

void _far _putimage(short *x*, **short** *y*, **char _huge ****image*, **short** *action* **);**

void _far _putimage_w(double *wx*, **double** *wy*, **char _huge ****image*, **short** *action* **);**

Arguments

x, y	Position of upper left corner of image (view coordinates)
wx, wy	Position of upper left corner of image (window coordinates)
image	Stored image buffer
action	Interaction with existing screen image (**_GAND**, **_GOR**, **_GXOR**, **_GPSET**, **_GPRESET**)

Returns Void

Graphics 2: Presentation Graphics Routines

The presentation graphics routines provide complete charting capabilities for line, bar, column, scatter, and pie charts.

Some charts plot both "categories," or non-numeric data such as time periods, and "values," or specific numeric data, such as sales. Presentation graphics routines support the following kinds of charts:

Chart Name	Description
Line	Category/value chart, with styles for lines between points and for no lines
Bar	Category/value chart, horizontal bars, styles for stacked and unstacked multiple series charts
Column	Category/value chart, vertical bars, with styles for stacked and unstacked multiple series charts
Scatter	Value/value chart, with styles for lines connecting points or for no lines
Pie	Pie chart, with optional percentages and exploded sections

The graphics package supports the IBM (and compatible) Enhanced Graphics Adapter (EGA), Color Graphics Adapter (CGA), certain operating modes of the Video Graphics Array (VGA) hardware configurations, and the Multicolor Graphics Array (MCGA). The graphics package also supports the Hercules Graphics Card, Graphics Card Plus, InColor Card, and 100-percent compatible cards, as well as the special Olivetti modes available on AT&T computers.

The **_pg_initchart** and **_pg_defaultchart** functions are generally used at the very beginning of a presentation graphics program.

The **_pg_chart** functions produce column charts, line charts, and bar charts. The **_pg_chartscatter** functions produce a scatter plot of data. The **_pg_chartpie** function generates a pie chart.

_pg_chart Generates a chart of the type specified in the *env* environment variable. It produces a column, bar, or line chart for a single series of data.

Include PGCHART.H

Prototype **short _far _pg_chart(chartenv _far*_env_,**
char _far* _far *_categories_, float _far*_values_, short _n_);

Arguments		
	env	Chart environment variable
	categories	Array of category variables
	values	Array of data values
	n	Number of data values to chart

Returns 0 if successful, nonzero if not

_pg_chartms Generates a multiple series of charts of the type specified in the *env* environment variable. It produces a column, bar, or line chart for a multiple series of data. All series must be the same length.

Include	PGCHART.H	
Prototype	**short _far _pg_chartms(chartenv _far ****env*, **char _far * _far****categories*, **float _far****values*, **short** *n*, **short** *nseries*, **short** *arraydim*, **char _far* _far****serieslabels* **);**	
Arguments	*env*	Chart environment variable
	categories	Array of category variables
	values	Two-dimensional array of data values (series, data)
	n	Number of data values to chart in a series
	nseries	Number of series to chart
	arraydim	Second (row) dimension of data array
	serieslabels	Array of labels for series
Returns	0 if successful, nonzero if not	

_pg_chartpie Generates a pie chart for a single series of data.

Include	PGCHART.H	
Prototype	**short _far _pg_chartpie(chartenv _far****env*, **char _far* _far****categories*, **float _far****values*, **short _far****explode*, **short** *n* **);**	
Arguments	*env*	Chart environment variable
	categories	Array of category names
	values	Array of data values
	explode	Array of explode flags; 1=explode, 0=do not explode
	n	Number of data values to chart
Returns	0 if successful, nonzero if not	

_pg_chartscatter Generates a scatter chart for a single series of data.

Include	PGCHART.H	
Prototype	**short _far _pg_chartscatter(chartenv _far ****env*, **float _far ****xvalues*, **float _far ****yvalues*, **short** *n* **);**	
Arguments	*env*	Chart environment variable
	xvalues	Array of *x*-axis data values
	yvalues	Array of *y*-axis data values
	n	Number of data values to chart
Returns	0 if successful, nonzero if not	

_pg_chartscatterms Generates a scatter chart for a multiple series of data.

Include	PGCHART.H
Prototype	**short _far _pg_chartscatterms(chartenv _far ****env*, **float _far****xvalues*, **float _far****yvalues*, **short** *nseries*, **short** *n*, **short** *rowdim*, **char _far* _far ****serieslabels* **);**

Arguments	*env*	Chart environment variable
	xvalues	Two-dimensional array of *x*-axis values
	yvalues	Two-dimensional array of *y*-axis values
	n	Number of data values to chart in a series
	nseries	Number of series to chart
	rowdim	Second (row) dimension of data array
	serieslabels	Array of labels for series
Returns	0 if successful, nonzero if not	

_pg_defaultchart Initializes all necessary variables in the chart environment for the specified default chart and chart style.

Include	PGCHART.H	
Prototype	**short _far _pg_defaultchart(chartenv _far** **env*, **short** *charttype*, **short** *chartstyle* **);**	
Arguments	*env*	Chart environment variable
	charttype	Chart type (**_PG_BARCHART, _PG_COLUMNCHART, _PG_LINECHART, _PG_SCATTERCHART, _PG_PIECHART**)
	chartstyle	Chart style
Returns	0 if successful, nonzero if not	

_pg_initchart Initializes chart line-style set, default palettes, screen modes, and character fonts. You must call this routine before any other charting routine.

Include	PGCHART.H
Prototype	**short _far _pg_initchart(void);**
Arguments	None
Returns	0 if successful, nonzero if not

Graphics 3: Font Display Routines

The font graphics routines display font-based characters on the screen.

The **_registerfonts** function initializes the fonts package with a set of disk-based type fonts. This must be done at the very beginning of any fonts program. The **_unregisterfonts** function frees fonts from memory when they are no longer needed.

The **_setfont** function makes a specified font the current active font for output. The **_outgtext** function displays text on the screen using the current font.

_getfontinfo Obtains the current font characteristics.

Include	GRAPH.H
Prototype	**short _far _getfontinfo(struct _fontinfo _far** ***fontbuffer*);**
Argument	*fontbuffer* Font information
Returns	Font information as a **_fontinfo** structure

_getgtextextent Determines the width of the specified text in the current font.

Include	GRAPH.H
Prototype	**short _far _getgtextextent(unsigned char _far *** *text* **);**
Argument	*text* Text to be analyzed
Returns	The width of the text in pixels

_outgtext Outputs text in the current font to the screen at the current position.

Include	GRAPH.H
Prototype	**void _far _outgtext(unsigned char _far** ***text*);**
Argument	*text* Text to be output
Returns	Void

_registerfonts Initializes the font library.

Include	GRAPH.H
Prototype	**short _far _registerfonts(unsigned char _far** ***filename*);**
Argument	*file name* File name of .FON files to register
* **Returns**	Void

_setfont Finds a single font that matches a specified set of characteristics and makes this font the current font.

Include	GRAPH.H
Prototype	**short _far _setfont(unsigned char _far** ***options*);**
Argument	*options* Font options string
* **Returns**	Void

_unregisterfonts Frees memory associated with fonts.

Include	GRAPH.H
Prototype	**void _far _unregisterfonts(void);**
Arguments	None
Returns	Void

***Errata: These functions do return values. See online help for details.**

Input and Output Routines

The input and output (I/O) routines of the standard C library allow you to read and write data to and from files and devices. In C, there are no predefined file structures; all data is treated as sequences of bytes.

Three types of I/O functions are available:

- Stream I/O, in which the data file is a stream of individual characters
- Low-level I/O, which uses the system's I/O capabilities directly
- Console and port I/O, which are stream routines for console or port

Stream I/O uses the **FILE** structure. The stream routines provide for buffered, formatted, or unformatted input and output.

Low-level I/O uses a file "handle" to access files. This handle is an integer value that is used to refer to the file in subsequent operations.

Do not mix stream and low-level routines on the same file or device.

Stream Routines

In the stream routines listed below, the following manifest constants are used:

- **EOF** is defined to be the value returned at end-of-file
- **NULL** is the null pointer
- **FILE** is the structure that maintains information about a stream
- **BUFSIZ** defines the default size of stream buffers, in bytes

clearerr Clears the error indicator for a stream.

Include	STDIO.H
Prototype	**void clearerr(FILE ****stream* **);**
Argument	*stream* Pointer to **FILE** structure
Returns	Void

fclose Closes a stream.

Include	STDIO.H
Prototype	**int fclose(FILE ****stream* **);**
Argument	*stream* Target **FILE** structure
Returns	0 if successful, **EOF** if not

feof Tests for end-of-file on a stream.

Include	STDIO.H
Prototype	**int feof(FILE** ***stream*);**
Argument	*stream* Pointer to **FILE** structure
Returns	A nonzero value when the current position is the end-of-file, 0 if not

ferror Tests for error on a stream.

Include	STDIO.H
Prototype	**int ferror(FILE** ***stream*);**
Argument	*stream* Pointer to **FILE** structure
Returns	A nonzero value to indicate an error in stream, 0 to indicate no error

fflush Flushes a stream.

Include	STDIO.H
Prototype	**int fflush(FILE** ***stream*);**
Argument	*stream* Pointer to **FILE** structure
Returns	0 if successful, if stream has no buffer, or if stream is open only for reading; returns **EOF** otherwise

fgetc Reads a character from a stream (function version).

Include	STDIO.H
Prototype	**int fgetc(FILE** ***stream*);**
Argument	*stream* Pointer to **FILE** structure
Returns	The character read; **EOF** may indicate error

fgetpos Gets the position indicator of a stream.

Include	STDIO.H
Prototype	**int fgetpos(FILE** ***stream*, **fpos_t** ***pos*);**
Arguments	*stream* Target stream *pos* Position indicator storage
Returns	0 if successful, a nonzero value if not **errno**: **EINVAL, EBADF**

fgets Reads a string from a stream.

Include	STDIO.H	
Prototype	**char *fgets(char ****string***, int** *n***, FILE ****stream* **);**	
Arguments	*string*	Storage location for data
	n	Number of characters stored
	stream	Pointer to **FILE** structure
Returns	A pointer to string if successful, **NULL** if unsuccessful or at end-of-file	

fopen Opens a stream.

Include	STDIO.H	
Prototype	**FILE *fopen(const char ****filename***, const char ****mode* **);**	
Arguments	*filename*	Path name of file
	mode	Type of access permitted such as **r**, **w**, **a**, **r+**, **w+**, **a+**, **t**, **b** (appended to type to indicate mode)
Returns	A pointer to the open file if successful, **NULL** if not	

fprintf Writes formatted data to a stream.

Include	STDIO.H	
Prototype	**int fprintf(FILE ****stream***, const char ****format* [[, *argument*]]... **);**	
Arguments	*stream*	Pointer to **FILE** structure
	format	Format-control string
Returns	The number of characters printed	

fputc Writes a character to a stream (function version).

Include	STDIO.H	
Prototype	**int fputc(int** *c***, FILE ****stream* **);**	
Arguments	*c*	Character to be written
	stream	Pointer to **FILE** structure
Returns	The character written; **EOF** may indicate error	

fputs Writes a string to a stream.

Include	STDIO.H	
Prototype	**int fputs(const char ****string***, FILE ****stream* **);**	
Arguments	*string*	String to be output
	stream	Pointer to **FILE** structure
Returns	0 if successful, nonzero if not	

fread Reads unformatted data from a stream.

Include STDIO.H

Prototype **size_t fread(void** ***buffer*, **size_t** *size*, **size_t** *count*, **FILE** ***stream* **);**

Arguments

buffer	Storage location for data
size	Item size in bytes
count	Maximum number of items to be read
stream	Pointer to **FILE** structure

Returns The number of items actually read

freopen Reassigns a FILE pointer.

Include STDIO.H

Prototype **FILE *freopen(const char** ***filename*, **const char** ***mode*, **FILE** ***stream* **);**

Arguments

filename	Path name of new file
mode	Type of access permitted such as **r, w, a, r+, w+, a+, t, b** (appended to type to indicate mode)
stream	Pointer to **FILE** structure

Returns A pointer to the newly opened file if successful, a **NULL** pointer if not

fscanf Reads formatted data from a stream.

Include STDIO.H

Prototype **int fscanf(FILE** ***stream*, **const char*** *format* [[, *argument*]] ... **);**

Arguments

stream	Pointer to **FILE** structure
format	Format-control string

Returns The number of fields successfully converted and assigned; **EOF** indicates an attempt to read the end-of-file

fseek Repositions FILE pointer to given location.

Include STDIO.H

Prototype **int fseek(FILE** ***stream*, **long** *offset*, **int** *origin* **);**

Arguments

stream	Pointer to **FILE** structure
offset	Number of bytes from origin
origin	Initial position (**SEEK_SET**, **SEEK_CUR**, **SEEK_END**)

Returns 0 if successful, a nonzero value if not

fsetpos Sets the position indicator of a stream.

Include STDIO.H

Prototype **int fsetpos(FILE** ***stream*, **const fpos_t** ***pos* **);**

Arguments

stream	Target stream
pos	Position-indicator storage

Returns 0 if successful, a nonzero value if not
errno: **EINVAL, EBADF**

ftell Gets current FILE pointer position.

Include STDIO.H

Prototype **long ftell(FILE *stream);**

Argument *stream* Target **FILE** structure

Returns The current position if successful, –1L if not
errno: **EINVAL, EBADF**

fwrite Writes unformatted data items to a stream.

Include STDIO.H

Prototype **size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);**

Arguments

buffer	Pointer to data to be written
size	Item size in bytes
count	Maximum number of items to be written
stream	Pointer to **FILE** structure

Returns The number of full items actually written

getc Reads a character from a stream (macro version).

Include STDIO.H

Prototype **int getc(FILE *stream);**

Argument *stream* Pointer to **FILE** structure

Returns The character read; **EOF** may indicate error

getchar Reads a character from stdin (macro version).

Include STDIO.H

Prototype **int getchar(void);**

Arguments None

Returns The character read; **EOF** may indicate error

gets Reads a line from stdin.

Include STDIO.H

Prototype **char *gets(char *buffer);**

Argument *buffer* Storage location for input string

Returns A pointer to its argument if successful, a **NULL** pointer if at end-of-file or unsuccessful

printf Writes formatted data to **stdout**.

Include	STDIO.H
Prototype	**int printf(const char ***_format_ **[[,** _argument_**]]...);**
Argument	_format_ Format-control string
Returns	The number of characters printed

putc Writes a character to a stream (macro version).

Include	STDIO.H
Prototype	**int putc(int** _c_**, FILE ***_stream_ **);**
Arguments	_c_ Character to be written _stream_ Pointer to **FILE** structure
Returns	The character written; **EOF** may indicate error

putchar Writes a character to **stdout** (macro version).

Include	STDIO.H
Prototype	**int putchar(int** _c_ **);**
Argument	_c_ Character to be written
Returns	The character written; **EOF** may indicate error

puts Writes a line to a stream.

Include	STDIO.H
Prototype	**int puts(const char ***_string_ **);**
Argument	_string_ String to be output
Returns	0 if successful, nonzero if not

rewind Repositions **FILE** pointer to beginning of a stream.

Include	STDIO.H
Prototype	**void rewind(FILE ***_stream_ **);**
Argument	_stream_ Pointer to **FILE** structure
Returns	Void

scanf Reads formatted data from **stdin**.

Include	STDIO.H
Prototype	**int scanf(const char ***_format_ **[[,** _argument_**]]...);**
Argument	_format_ Format control string
Returns	The number of fields converted and assigned if successful, 0 if no fields were assigned, **EOF** for an attempt to read end-of-file

sprintf Writes formatted data to string.

Include	STDIO.H
Prototype	**int sprintf(char **buffer*, const char **format* [[, *argument*]] ...);**
Arguments	*buffer* Storage location for output *format* Format-control string
Returns	The number of characters stored in buffer

sscanf Reads formatted data from string.

Include	STDIO.H
Prototype	**int sscanf(const char **buffer*, const char **format* [[, *argument*]] ...);**
Arguments	*buffer* Stored data *format* Format-control string
Returns	The number of fields converted and assigned if successful, 0 if no fields were assigned, **EOF** for an attempt to read at end-of-string

tmpfile Creates a temporary file.

Include	STDIO.H
Prototype	**FILE *tmpfile(void);**
Arguments	None
Returns	A stream pointer if successful, **NULL** if not

tmpnam Generates a temporary file name.

Include	STDIO.H
Prototype	**char *tmpnam(char **string*);**
Argument	*string* Pointer to temporary name
Returns	A pointer to the new name if successful, **NULL** if not

ungetc Places a character in the input stream buffer.

Include	STDIO.H
Prototype	**int ungetc(int *c*, FILE **stream*);**
Arguments	*c* Character to be pushed *stream* Pointer to **FILE** structure
Returns	The character argument *c* if successful, **EOF** if not

Low-Level Routines

The low-level input and output calls do not buffer or format data.

Files opened by low-level calls are referenced by a "file handle," an integer value used by the operating system to refer to the file.

close Closes a file.

Include	IO.H
Prototype	**int close(int** *handle* **);**
Argument	*handle* Handle referring to open file
Returns	0 if successful, –1 if not **errno: EBADF**

creat Creates a file.

Include	IO.H, SYS\TYPES.H, SYS\STAT.H
Prototype	**int creat(char** **filename*, **int** *pmode* **);**
Arguments	*filename* Path name of new file *pmode* Permission setting (**S_IWRITE** , **S_IREAD**, **S_IREAD | S_IWRITE**)
Returns	A handle if successful, –1 if not **errno: EACCES, EMFILE, ENOENT**

eof Tests for end-of-file.

Include	IO.H
Prototype	**int eof(int** *handle* **);**
Argument	*handle* Handle referring to open file
Returns	1 if the current position is the end-of-file and 0 if it is not, –1 to indicate an error **errno: EBADF**

lseek Repositions file pointer to a given location.

Include	IO.H, STDIO.H
Prototype	**long lseek(int** *handle*, **long** *offset*, **int** *origin* **);**
Arguments	*handle* Handle referring to open file *offset* Number of bytes from origin *origin* Initial position (**SEEK_SET**, **SEEK_CUR**, **SEEK_END**)
Returns	The new position offset (in bytes) from the beginning of the file if successful, –1L if not **errno: EBADF, EINVAL**

open Opens a file.

Include	FCNTL.H, IO.H, SYS\TYPES.H, SYS\STAT.H
Prototype	**int open(char** **path*, **int** *oflag* **[[, int** *pmode***]]);**

Arguments	*path*	File path name
	oflag	Type of operations allowed such as **O_APPEND, O_BINARY, O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_TEXT, O_TRUNC, O_WRONLY** (may be joined by |)
	pmode	Permission setting (**S_IWRITE, S_IREAD, S_IREAD | S_IWRITE**)
Returns	A handle if successful, –1 if not **errno: EACCES, EEXIST, EMFILE, ENOENT**	

read Reads data from a file.

Include	IO.H	
Prototype	**int read(int** *handle*, **char** **buffer*, **unsigned int** *count* **);**	
Arguments	*handle*	Handle referring to open file
	buffer	Storage location of data
	count	Maximum number of bytes
Returns	The number of bytes actually read or 0 at end-of-file if successful; –1 if not **errno: EBADF**	

tell Gets current file-pointer position.

Include	IO.H	
Prototype	**long tell(int** *handle* **);**	
Argument	*handle*	Handle referring to open file
Returns	The current position if successful, –1L if not **errno: EBADF**	

write Writes data to a file.

Include	IO.H	
Prototype	**int write(int** *handle*, **void** **buffer*, **unsigned int** *count* **);**	
Arguments	*handle*	Handle referring to open file
	buffer	Data to be written
	count	Number of bytes
Returns	The number of bytes actually written if successful, –1 if not **errno: EBADF, ENOSPC**	

Console and Port I/O Routines

The console and port I/O routines perform reading and writing operations on your console or on the specified port.

The **cgets, cscanf, getch, getche,** and **kbhit** routines take input from the console.

The **cprintf, cputs, putch,** and **ungetch** routines write to the console.

The console or port does not have to be opened or closed before I/O is performed.

The console I/O routines use the corresponding MS-DOS system calls to read and write characters. Since these routines are not compatible with stream or low-level library routines, console routines should not be used with them.

cgets
Reads a string from the console.

Include	CONIO.H
Prototype	**char *cgets(char** ***buffer*** **);**
Argument	*buffer* Storage location for data
Returns	A pointer to the start of the string, which is at *str*[2]

cprintf
Writes formatted data to the console.

Include	CONIO.H
Prototype	**int cprintf(char** ***format*** **[[,** ***argument*****]] ...);**
Argument	*format* Format-control string
Returns	The number of characters printed

cputs
Writes a string to the console.

Include	CONIO.H
Prototype	**int cputs(char** ***string*** **);**
Argument	*string* Output string
Returns	0 if successful, nonzero if not

cscanf
Reads formatted data from the console.

Include	CONIO.H
Prototype	**int cscanf(char** ***format*** **[[,** ***argument*****]]**...);
Argument	*format* Format-control string
Returns	The number of fields converted and assigned if successful (0 means no fields were assigned), **EOF** for an attempt to read end-of-file

getch
Reads a character from the console.

Include	CONIO.H
Prototype	**int getch(void);**
Arguments	None
Returns	The character read

getche
Reads a character from the console and echoes it.

Include	CONIO.H
Prototype	**int getche(void);**
Arguments	None
Returns	The character read

kbhit
Checks for a keystroke at the console.

Include	CONIO.H
Prototype	**int kbhit(void);**
Arguments	None
Returns	A nonzero value if a key has been pressed, 0 if not

putch
Writes a character to the console.

Include	CONIO.H	
Prototype	**int putch(int** *c* **);**	
Argument	*c*	Character to be output
Returns	The argument *c* if successful, **EOF** if not	

ungetch
"Ungets" the last character read from the console so that it becomes the next character read.

Include	CONIO.H	
Prototype	**int ungetch(int** *c* **);**	
Argument	*c*	Character to be pushed
Returns	The argument *c* if successful, **EOF** if not	

Math Routines

The math routines allow you to perform common mathematical calculations.

All math routines work with floating-point values and therefore require floating-point support.

abs, fabs, labs
The **abs**, **fabs**, and **labs** routines return the absolute value of an integer, a double, and a long argument, respectively.

Includes	STDLIB.H (**abs**, **labs**), MATH.H (**fabs**)

Prototypes	**int abs(int** *n* **);**	
	double fabs(double *x* **);**	
	long labs(long *x* **);**	
Arguments	*n*	Integer (**abs**) or long (**labs**) value
	x	Floating-point value
Returns	Absolute value of its argument	

acos Calculates the arccosine.

Includes	FLOAT.H, MATH.H	
Prototype	**double acos(double** *x* **);**	
Argument	*x*	Value whose arccosine is to be calculated
Returns	The arccosine result if successful, or 0 if $x > 1$ **errno: EDOM**	

asin Calculates the arcsine.

Includes	FLOAT.H, MATH.H	
Prototype	**double asin(double** *x* **);**	
Argument	*x*	Value whose arcsine is to be calculated
Returns	The arcsine result if successful, or 0 if $x >1$ **errno: EDOM**	

atan, atan2 Calculates the arctangent of *x* (**atan**) or the arctangent of *y*/*x* (atan2).

Includes	FLOAT.H, MATH.H	
Prototypes	**double atan(double** *x* **);**	
	double atan2(double *y*, **double** *x* **);**	
Argument	*x*, *y*	Floating-point values
Returns	**atan**: the arctangent result **atan2**: the arctangent of *y*/*x*, or 0 if both arguments are 0 **errno: EDOM**	

ceil Rounds the argument up to an integer.

Includes	FLOAT.H, MATH.H	
Prototype	**double ceil(double** *x* **);**	
Argument	*x*	Floating-point value
Returns	The double result	

cos, cosh

Calculates the cosine (**cos**) or the hyperbolic cosine (**cosh**).

Includes FLOAT.H, MATH.H

Prototypes **double cos(double** *x* **);**

double cosh(double *x* **);**

Argument *x* Angle (in radians)

Returns **cos**: the cosine result if successful, 0 if not
cosh: the hyperbolic result if successful, or **HUGE_VAL** if the result is too large
errno: ERANGE

exp

Calculates the exponential function.

Includes FLOAT.H, MATH.H

Prototype **double exp(double** *x* **);**

Argument *x* Floating-point value

Returns The exponential value if successful, **HUGE_VAL** on overflow, 0 on underflow
errno: ERANGE

floor

Rounds the argument down to an integer.

Includes FLOAT.H, MATH.H

Prototype **double floor(double** *x* **);**

Argument *x* Floating-point value

Returns The floating-point result

fmod

Finds the floating-point remainder.

Includes FLOAT.H, MATH.H

Prototype **double fmod(double** *x*, **double** *y* **);**

Argument *x*, *y* Floating-point values

Returns The floating-point remainder, or 0 if *y* is 0

frexp

Calculates an exponential value.

Includes FLOAT.H, MATH.H

Prototype **double frexp(double** *x*, **int** **expptr* **);**

Argument *x* Floating-point value
expptr Pointer to stored integer exponent

Returns The mantissa, or 0 if *x* is 0

ldexp Calculates the argument times 2^{exp}.

Includes	FLOAT.H, MATH.H
Prototype	**double ldexp(double** *x*, **int** *exp* **);**
Arguments	*x* Floating-point value *exp* Integer exponent
Returns	An exponential value if successful, **HUGE_VAL** on overflow **errno: ERANGE**

log, log10 Calculates the natural logarithm (**log**) or the base-10 log (**log10**).

Includes	FLOAT.H, MATH.H
Prototypes	**double log(double** *x* **);** **double log10(double** *x* **);**
Argument	*x* Floating-point value
Returns	A logarithm result if successful, **–HUGE_VAL** if not **errno: EDOM** (if $x < 0$), **ERANGE** (if $x = 0$)

modf Breaks argument into integer and fractional parts.

Includes	FLOAT.H, MATH.H
Prototype	**double modf(double** *x*, **double** **intptr* **);**
Arguments	*x* Floating-point value *intptr* Pointer to stored integer position
Returns	The signed fractional portion of *x*

pow Calculates a value raised to a power.

Includes	FLOAT.H, MATH.H
Prototype	**double pow(double** *x*, **double** *y* **);**
Arguments	*x* Number to be raised *y* Power of *x*
Returns	The argument *x* raised to the *y* power if successful, **HUGE_VAL** if not

rand, srand The **rand** function returns a pseudorandom integer in the range 0–32,767. The **srand** function initializes the random number generator.

Include	STDLIB.H
Prototypes	**int rand(void);** **void srand(unsigned** *seed* **);**
Argument	*seed* Seed for random-number generation (**srand**)
Returns	**rand**: a pseudorandom number **srand**: void

sin, sinh Calculates the sine (**sin**) or hyperbolic sine (**sinh**).

Includes FLOAT.H, MATH.H

Prototypes **double sin(double** *x* **);**

double sinh(double *x* **);**

Argument *x* Angle (in radians)

Returns **sin**: the sine of *x* if successful, 0 if not
sinh: the hyperbolic sine of *x* if successful, **HUGE_VAL** if not
errno: **ERANGE**

sqrt Finds the square root.

Includes FLOAT.H, MATH.H

Prototype **double sqrt(double** *x* **);**

Argument *x* Nonnegative floating-point value

Returns A square root if successful, 0 if not
errno: **EDOM**

tan, tanh Calculates the tangent (**tan**) or hyperbolic tangent (**tanh**).

Includes FLOAT.H, MATH.H

Prototypes **double tan(double** *x* **);**

double tanh(double *x* **);**

Argument *x* Angle (in radians)

Returns **tan:** the tangent of *x* if successful, 0 if not
tanh: the hyperbolic tangent of *x*
errno: **ERANGE** (**tan** only)

Memory-Allocation Routines

The memory-allocation routines allocate, free, analyze, and reallocate blocks of memory.

Many of the memory-allocation functions are prefixed by an **_f** or an **_n**. This notation means use the far (**_f**) heap or the near (**_n**) heap.

The **malloc** family of routines (**malloc**, **_fmalloc**, and **_nmalloc**) allocates memory blocks of a specified size. The **calloc** function allocates storage for an array. The **halloc** function allocates storage for a huge array.

The **realloc** routine changes the size of an allocated block.

calloc Allocates storage for an array.

Includes	MALLOC.H, STDLIB.H
Prototype	**void *calloc(size_t** *num*, **size_t** *size* **);**
Arguments	*num* Number of elements *size* Length in bytes of each element
Returns	A void pointer to the allocated space if successful, **NULL** if not

free, _ffree, hfree, _nfree Frees a block of memory previously allocated by the corresponding **malloc** routine. The corresponding routines are listed below:

Free Function	Allocation Function
_ffree	**_fmalloc**
free	**calloc, malloc, realloc**
hfree	**halloc**
_nfree	**_nmalloc**

Includes	MALLOC.H, STDLIB.H (ANSI-compatible **free** only)
Prototypes	**void _ffree(void _far** **memblock* **);** **void free(void** **memblock* **);** **void _nfree(void near** **memblock* **);** **void hfree(void huge** **memblock* **);**
Argument	*memblock* Allocated memory block
Returns	Void

malloc , _fmalloc, _nmalloc Allocates a block of memory. The **_fmalloc** function allocates the block in the far heap. The **_nmalloc** function allocates the block in the near heap.

Includes	MALLOC.H, STDLIB.H (ANSI-compatible **malloc** only)
Prototypes	**void *malloc(size_t** *size* **);** **void _far *_fmalloc(size_t** *size* **);** **void near *_nmalloc(size_t** *size* **);**
Argument	*size* Bytes to allocate
Returns	A void pointer to the allocated space if successful, **NULL** if not

realloc Reallocates a block.

Include	MALLOC.H, STDLIB.H	
Prototype	**void *realloc(void ****memblock***, size_t** *size* **);**	
Arguments	*memblock*	Pointer to previously allocated memory block
	size	New size in bytes
Returns	A pointer to the reallocated memory if successful, **NULL** if not	

Process-Control Routines

The term "process" refers to a program being executed by the operating system.

Use the process-control routines to

- Terminate a process (**abort**, **exit**, and **_exit**)
- Call a new function when a process terminates (**atexit**)
- Start a new process (**system**)

Use the **abort** and **_exit** functions to exit without flushing stream buffers. Use the **exit** function to exit after flushing stream buffers.

Use the **atexit** function to create a list of functions to be executed when the calling program exits.

Use the **system** call to execute a given MS-DOS command.

abort Aborts a process.

Include	PROCESS.H or STDLIB.H
Prototype	**void abort(void);**
Arguments	None
Returns	Void

atexit Executes functions at program termination.

Include	STDLIB.H	
Prototype	**int atexit(void (****func***)(void));**	
Argument	*func*	Function to be called
Returns	A pointer to *func* if successful, **NULL** if not	

exit, _exit
Terminates the process after flushing buffers (**exit**); terminates the process without flushing buffers (**_exit**).

Include	PROCESS.H or STDLIB.H	
Prototypes	**void exit(int** *status* **);**	
	void _exit(int *status* **);**	
Argument	*status*	Exit status
Returns	Void	

system
Executes an MS-DOS command.

Include	PROCESS.H, STDLIB.H	
Prototype	**int system(const char** **command* **);**	
Argument	*command*	Command to be executed
Returns	0 if successful, –1 if not **errno**: **E2BIG, ENOENT, ENOEXEC, ENOMEM**	

Searching and Sorting Routines

The **bsearch**, **lfind**, **lsearch**, and **qsort** routines provide helpful binary-search, linear-search, and quick-sort utilities.

bsearch
Performs a binary search.

Includes	STDLIB.H, SEARCH.H	
Prototype	**void *bsearch(const void** **key*, **const void** **base*, **size_t** *num*, **size_t** *width*, **int(** **compare* **)(const void** **elem1*, **const void** **elem2* **));**	
Arguments	*key*	Object to search for
	base	Pointer to base of search data
	num	Number of elements
	width	Width of elements
	compare	Compare function
	elem1, *elem2*	Array elements to compare
Returns	A pointer if successful, **NULL** if not	

lfind, lsearch
Performs a linear search for given value. If the value is not found, **lsearch** adds it to the end of the list.

Includes	STDLIB.H, SEARCH.H

Prototypes	**char *lfind(char ****key*, **char ****base*, **unsigned ****num*, **unsigned** *width*, **int(****compare* **)(const void ****elem1*, **const void ****elem2* **));**	
	char *lsearch(const char ****key*, **const char ****base*, **unsigned ****num*, **unsigned *width*, **int(****compare* **) (const void ****elem1*, **const void ****elem2* **));**	
Arguments	*key*	Object to search for
	base	Pointer to base of search data
	num	Number of elements
	width	Width of elements
	compare	Compare function
	elem1, *elem2*	Array elements to compare
Returns	A pointer if successful, NULL if not	

qsort Performs a quick sort.

Includes	STDLIB.H, SEARCH.H	
Prototype	**void qsort(void ****base*, **size_t** *num*, **size_t** *width*, **int(****compare* **)(const void ****elem1*, **const void ****elem2* **));**	
Arguments	*base*	Start of target array
	num	Array size in elements
	width	Element size in bytes
	compare	Compare function
	elem1, *elem2*	Array elements to compare
Returns	Void	

String-Manipulation Routines

A wide variety of string routines is available in the run-time library. With these functions, you can do the following:

- Copy strings (**strcat**, **strcpy**, **strdup**, **strncat**, **strncpy**)
- Search for strings, individual characters, or characters from a given set (**strchr**, **strcspn**, **strpbrk**, **strrchr**, **strspn**, **strstr**)
- Perform string comparisons (**strcmp**, **strcmpi**, **stricmp**, **strncmp**, **strnicmp**)
- Find the length of a string (**strlen**)
- Convert strings to a different case (**strlwr**, **strupr**)
- Set characters of the string to a given character (**strnset**, **strset**)
- Break strings into tokens (**strtok**)

All string functions work on null-terminated character strings.

Use the buffer-manipulation routines described earlier in this appendix for manipulating character arrays that do not end with a null character.

strcat, strcpy, strdup, strncat, strncpy Use these routines to copy and concatenate strings. The list below describes each function.

Function	Action
strcat	Append (concatenate) a string
strcpy	Copy one string to another
strdup	Duplicate a string
strncat	Append a specified number of characters to a string
strncpy	Copy a specified number of characters from one string to another

Include STRING.H

Prototypes **char *strcat(char ***_dest_**, const char ***_src_ **);**

char *strcpy(char *_dest_, **const char ***_src_);

char *strdup(const char *_string_ **);**

char *strncat(char ***_dest_, const char ***_src_**, size_t** _n_ **);**

char *strncpy(char ***_dest_, const char ***_src_**, size_t** _n_ **);**

Arguments

dest	Destination string
src	Source string
string	Null-terminated string
_n_	Number of characters

Returns

strcat: a pointer to the concatenated string
strcpy: _dest_ string
strdup: a pointer if successful, NULL if not
strncat, strncpy: a pointer to _dest_ string

strchr, strcspn, strpbrk, strrchr, strspn, strstr Use these routines to search strings. The list below describes each function.

Function	Action
strchr	Finds first occurrence of a given character in a string
strcspn	Finds first occurrence of a character from a given character set in a string
strpbrk	Finds first occurrence of a character from one string in another
strrchr	Finds last occurrence of a given character in a string
strspn	Finds first substring from a given character set in a string
strstr	Finds first occurrence of a given string in another string

Include STRING.H

Prototypes

char *strchr(const char ****string*, int** *c* **);**

size_t strcspn(const char ****string1*, const char ****string2* **);**

char *strpbrk(const char ****string1*, const char ****string2* **);**

char *strrchr(const char ****string*, int** *c* **);**

size_t strspn(const char ****string1*, const char ****string2* **);**

char *strstr(const char ****string1*, const char ****string2* **);**

Arguments

string, string1, string2	Null-terminated strings
c	Character

Returns

strchr: a pointer if successful, **NULL** if not
strcspn: an offset into *string1*
strpbrk: a pointer to the first matching character in *string1*, **NULL** if no match is found
strrchr: a pointer to the last occurrence of *c* in *string*, **NULL** if *c* is not found
strspn: the position of the first nonmatching character in *string1*
strstr: a pointer to the first occurrence of *string2* in *string1*, or **NULL** if *string2* is not found

strcmp, strcmpi, stricmp, strncmp, strnicmp Use these routines to compare strings. The list below describes the operation of each function. An "**n**" in the function name means to use up to **n** characters; "**i**" in the name means to operate without regard to the case of the string.

Function	Action
strcmp	Compares two strings
strcmpi	Compares two strings without regard to case ("**i**" indicates that this function is case insensitive)
stricmp	Compares two strings without regard to case (identical to **strcmpi**)
strncmp	Compares characters of two strings
strnicmp	Compares characters of two strings without regard to case

Include STRING.H

Prototypes

int strcmp(const char ******string1*, **const char** ******string2* **);**

int strcmpi(const char ******string1*, **const char** ******string2* **);**

int stricmp(const char ******string1*, **const char** ******string2* **);**

int strncmp(const char ******string1*, **const char** ******string2*, **size_t** *n* **);**

int strnicmp(const char ******string1*, **const char** ******string2*, **size_t** *n* **);**

Arguments

string1	Destination string
string2	Source string
n	Number of characters

Returns A negative value if *string1*<*string2*, 0 if *string1* = *string2*, a positive value if *string1*>*string2*

strlen The **strlen** function returns the length in bytes of the string, not including the terminating null character (**\0**).

Include STRING.H

Prototype **size_t strlen(const char** ******string* **);**

Argument *string* Null-terminated string

Returns The string length

strlwr, strupr

The **strlwr** and **strupr** routines convert the characters of a string to lowercase and uppercase, respectively.

Include	STRING.H
Prototypes	**char *strlwr(char ****string* **);**
	char *strupr(char ****string* **);
Argument	*string* String to be converted
Returns	A pointer to a copy of the converted input string

strnset, strset

The routines **strnset** and **strset** set the characters of a string to a specified character. The **strnset** function sets the first *n* characters in the string to the specified character. The **strset** function sets the entire string to the specified character.

Include	STRING.H
Prototypes	**char *strnset(char ****string***, int** *c***, size_t** *n* **);**
	char*strset(char ****string*, int** *c* **);**
Arguments	*string* String to be set *c* Character setting *n* Number of characters set
Returns	A pointer to *string*

strtok

The **strtok** function finds a token in a string. A "token" is a series of characters delimited by a character from a specified set. For example, use the **strtok** function to break an input line into the component words.

Include	STRING.H
Prototype	**char *strtok(char ****string1***, const char ****string2* **);**
Arguments	*string1* String containing tokens *string2* Set of delimiter characters
Returns	A pointer to a token in *string1*

Time Routines

Use the time routines to get the current time, convert it to a convenient format, and store it according to your particular needs.

The current time is always taken from the system time.

The **time** function returns the current time as the number of seconds elapsed since Greenwich mean time, January 1, 1970.

Use the **asctime**, **ctime**, **gmtime**, and **mktime** functions to manipulate the time value.

asctime Converts a time from a structure to a character string.

Include	TIME.H
Prototype	**char *asctime(const struct tm ****timeptr* **);**
Argument	*timeptr* Time structure
Returns	A pointer to string result

clock Returns the elapsed CPU time for a process.

Include	TIME.H
Prototype	**clock_t clock(void);**
Arguments	None
Returns	The elapsed processor time if successful, –1 if not

ctime Converts time from a long integer to a character string.

Include	TIME.H
Prototype	**char *ctime(const time_t ****timer* **);**
Argument	*timer* Pointer to stored time
Returns	A pointer to string result; NULL if time represents a date before 1980

difftime Computes the difference between two times.

Include	TIME.H
Prototype	**double difftime(time_t** *timer1***, time_t** *timer0* **);**
Arguments	*timer0*, *timer1* Beginning and ending times
Returns	The difference in elapsed time between *timer1* and *timer0*

ftime Gets current system time as structure.

Includes	SYS\TYPES.H, SYS\TIMEB.H
Prototype	**void ftime(struct timeb ****timeptr* **);**
Argument	*timeptr* Pointer to time structure
Returns	Void

gmtime Converts time from integer to structure.

Include	TIME.H
Prototype	**struct tm *gmtime(const time_t ****timer* **);**
Argument	*timer* Pointer to stored time
Returns	A pointer to a structure

mktime Converts time to a calendar value.

Include	TIME.H
Prototype	**time_t mktime(struct tm** ***timeptr*);**
Argument	*timeptr* Local time structure
Returns	The encoded calendar time if successful, –1 if not

time Gets current system time as a long integer.

Include	TIME.H
Prototype	**time_t time(time_t** ***timer*);**
Argument	*timer* Storage location for time
Returns	The elapsed time

Glossary

8087 or 80287 coprocessor: Intel hardware products that provide very fast and precise floating-point number processing.

aggregate types: Arrays, structures, and unions.

ANSI (American National Standards Institute): The national institute responsible for defining programming-language standards to promote portability of these languages between different computer systems. The ANSI standard for C will become official in 1990.

argc: The traditional name for the first argument to the **main** function in a C source program. It is an integer that specifies how many arguments are passed to the program from the command line.

argument: A value passed to a function.

argv: The traditional name for the second argument to the **main** function in a C source program. It is a pointer to an array of strings. Traditionally, the first string is the program name, and each following string is an argument passed to the program from the command line.

array: A set of elements with the same type.

array pointer: A pointer that holds the address of any element of an array.

ASCII (American Standard Code for Information Interchange): A set of 256 codes that many computers use to represent letters, digits, special characters, and other symbols. Only the first 128 of these codes are standardized; the remaining 128 are special characters that are defined by the computer manufacturer.

automatic variable: A variable, declared in a block, whose value is discarded when the program exits from the block. See "static variable" and "lifetime."

background color: A long integer representing the background color of the display screen. In graphics modes, the background color applies to the entire screen. In text modes, the background color specifies the text background for each character. See "foreground color."

basic data types: The integral, enumerated, floating-point, and pointer types in the C language.

binary file: A file that is not used for text processing. It may be an executable file, a data file, or some other nontext file.

binary format: A method of data representation in which data are stored directly from memory to disk with no translations. In binary format, numeric values are stored as binary numbers and are not translated to ASCII characters.

binary mode: A method of accessing files in which no translations are performed. There is no specific end-of-file character.

binary operator: An operator that takes two operands. Binary operators in the C language are the multiplicative operators (*/), additive operators (+ –), shift operators (<< >>), relational operators (< > <= >= == !=), bitwise operators (& | ^), logical operators (&& ||), and the sequential-evaluation operator (,).

bit: A binary digit (either 0 or 1), the smallest unit of information used with computers. Eight bits make up one byte.

bit field: A type of structure that allows manipulation of individual bits or groups of bits.

bit-mapped font: A font in which each character is defined by (mapped to) the bits of an array.

bitwise operator: An operator used to manipulate bits in an integer expression. Bitwise operators in the C language are & (AND), | (inclusive OR), ^ (exclusive OR), << (left shift), >> (right shift), and ~ (one's complement).

block: A sequence of declarations, definitions, and statements enclosed within curly braces ({ }).

bounding rectangle: An imaginary rectangle that defines the outer limits of a rounded shape such as an ellipse, arc, or pie.

byte: The unit of measure used for computer memory and data storage. One byte contains eight bits and can store one ASCII character.

case label: The **case** keyword and the constant, or constant expression, that follows it.

CGA: IBM's Color Graphics Adapter.

character code: A numeric code that represents a character. The default ASCII character set used in all PCs and PS/2s comprises 256 eight-bit character codes.

character constant: A character enclosed in single quotes, for example, `'p'`. A character constant has a type of **char**. See "string constant."

character set: A set of alphabetic and numeric characters and symbols.

clipping: The process of determining which parts of a graphics image lie within the clipping region. Parts of the image that lie outside this region are "clipped"; that is, they are not displayed.

clipping region: The rectangular area of the screen where graphics display occurs.

color index: A short integer that represents a displayable color. See "remapping" and "color value."

color value: A long integer representing an absolute color. See "remapping" and "color index."

command-line argument: A value passed to a program when the program begins execution.

conditional expression: An expression consisting of three operands joined by the ternary (? :) operator. Similar to an **if-else** construct, a conditional expression is used to evaluate either of two expressions depending on the value of a third expression.

constant expression: An expression that evaluates to a constant. A constant expression may involve integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant expressions.

current color: The color index for the color in which graphics pixels are displayed. The current color can be examined with **_getcolor** or changed with **_setcolor**.

declaration: A construct that associates the name and the attributes of a variable, function, or type.

default: A condition that is assumed by a program if not specified.

definition: A construct that initializes and allocates storage for a variable or that specifies the name, formal parameters, body, and return type of a function.

dimension: The number of subscripts required to specify a single array element.

directive: An instruction to the C preprocessor to perform an action on source-program text before compilation.

double precision: A real (floating-point) value that occupies eight bytes of memory. Double precision values are accurate to 15 or 16 digits.

EGA: Enhanced Graphics Adapter.

enumeration type: A user-defined data type with values that range over a set of named integral constants.

escape sequence: A specific combination of a backslash (\) followed by a letter or combination of digits, which represents white space and nonprinting characters within strings and character constants.

expression: A combination of operands and operators that yields a single value.

external variable: A variable that is defined outside any function in a C source file and is used in other source files in the same program.

file handle: An integer value that is returned when a library function that performs low-level input/output opens or creates a file. The file handle is used to refer to that file in later operations.

file pointer: A value that keeps track of the current position in an input or output stream. It is updated to reflect the new position each time a read or write operation takes place.

FILE pointer: A pointer to a structure of type FILE that contains information about a file. It is returned by library functions that create or open files and use stream input/output.

fill flag: A parameter that determines whether a shape will be drawn as a solid.

fill mask: A group of pixels that defines the pattern used to fill a graphics shape.

fill pattern: The design defined by the fill mask and used to fill a shape.

font: A description of the style and shapes of the characters in a character set.

foreground color: The color index for the color in which text is displayed. See "background color."

format specification: A string that specifies how the **printf** and **scanf** families of functions interpret input and output data.

function: A collection of declarations and statements that has a unique name and can return a value.

function body: A statement block containing the local variable declarations and statements of a function.

function call: An expression that passes control and arguments (if any) to a function.

function declaration: A declaration that states the name, return type, and storage class of a function that is defined explicitly elsewhere in the program.

function definition: A definition that specifies a function's name, its formal parameters, the declarations and statements that define what it does, and (optionally) its return type and storage class.

function pointer: A pointer that holds the address of a function.

function prototype: A function declaration that includes a list of the names and types of formal parameters in the parentheses following the function name.

global: See "visibility."

graphics mode: See "video mode."

header file: An external source file that contains commonly used declarations and definitions. The **#include** directive is used to insert the contents of a header file into a C source file.

hexadecimal: The base-16 numbering system whose digits are 0 through F. The letters A through F represent the decimal numbers 10 through 15. It is often used in computer programming because it is easily converted to and from binary, the base-2 numbering system the computer itself uses.

HGC: Hercules monochrome Graphics Card.

identifier: A user-defined name in a C program. Identifiers name variables, functions, macros, constants, and data types.

include file: See "header file."

Incolor Card: Hercules InColor Card, a 16-color version of the HGC+.

indirection: Accessing a data object through a pointer, rather than directly by name.

initialize: To assign a value to a variable, often at the time the variable is declared.

in-line assembler: The part of QuickC that converts assembly-language instructions into machine language.

in-line assembly code: Assembly language instructions that appear within a QuickC source program.

input/output: The processes involved in reading (input) and writing (output) data.

integer: A whole number represented in the machine as a 16-bit two's-complement binary number. A signed integer has a range of –32,768 to 32,767. An unsigned integer has a range of 0 to 65,535. See "long integer."

I/O: Abbreviation for input/output.

keyword: A word with a special, predefined meaning for the C compiler.

label: A unique name followed by a colon. Labels are used to denote statements to which a **goto** statement can branch. See "**case** label."

library: A file containing compiled modules. The linker extracts modules from the library file and combines them with the user-created object file to form an executable program.

lifetime: The time, during program execution, that a variable or function exists. An "automatic" variable has storage and a defined value only in the block where it is defined or declared. A "static" variable exists for the duration of the program.

line style: An unsigned short integer (16 bits) that specifies the pattern with which lines will be drawn. Each bit specifies whether a corresponding pixel in the line will be displayed. The default line style is a solid line.

local: See "visibility."

long integer: A whole number represented inside the machine as a 32-bit two's-complement binary number. A signed long integer has a range of –2,147,483,648 to 2,147,483,647. An unsigned long integer has a range of 0 to 4,294,967,295. See "integer."

low-level input and output routines: Run-time library routines that perform unbuffered, unformatted I/O operations, for example, **creat**, **read**, **write**, and **lseek**.

lvalue: An expression (such as a variable name) that refers to a memory location and is required as the left-hand operand of an assignment operation, or as the single operand of a unary operator.

machine language: A series of binary numbers that a computer executes as program instructions.

macro: An identifier defined in a **#define** preprocessor directive to represent another series of characters.

main function: The function with which program execution begins (the program's entry point).

manifest constant: See "symbolic constant."

MCGA (Multicolor Graphics Array): The video subsystem integrated into the PS/2 Model 30. Also, Memory Controller Gate Array, one of the components of the Model 30's video subsystem.

member: One of the elements of a structure or union.

member-of operator: The dot operator (.), which is used with the name of a structure and one or more fields to identify a structure member.

mode: See "video mode."

monochrome display: A computer monitor capable of showing only two colors—black and a second color such as white, green, or amber. Some monochrome monitors can also show the second color with higher intensity or with underlined text.

Monochrome Display Adapter (MDA): A printed-circuit card that controls the display and can show text only at medium resolution in one color.

newline character: The character used to mark the end of a line in a text file, or the escape sequence (**\n**) used to represent this character.

null character: The ASCII character encoded as the value 0, represented as the escape sequence (**\0**) in a source file. A null character marks the end of a string.

null pointer: A pointer to nothing, expressed as the value 0.

one's complement: The arithmetic operation in which all 1 bits are converted to 0 bits and vice versa. The tilde character (~) is the one's-complement operator.

operand: A constant or variable value that is manipulated in an expression.

operator: One or more symbols that specify how the operand or operands of an expression are manipulated. See "unary operator," "binary operator," and "ternary operator."

origin: The point on the screen at which the *x* and *y* coordinates are both equal to 0. On the physical screen, the origin is at the upper left corner.

palette: The displayable colors for a given video mode. The CGA modes operate with a set of predetermined palette colors. The EGA, VGA, and MCGA color modes operate with a redefinable palette of colors.

parameter: An identifier that receives a value passed to a function.

path: The name that defines the location of a file or directory. A path may include a drive name and one or more directory names.

PGA (Professional Graphics Adapter): Another name for IBM's PGC.

physical coordinates: The coordinate system defined by the hardware. The physical coordinate system has the origin (0, 0) at the upper left corner of the screen. The value of *x* increases from left to right, and the value of *y* increases from top to bottom. See "viewport coordinates."

pixel: A single dot on the screen. It is the smallest item that may be manipulated with the graphics library, and it is the basic unit of the viewport-coordinate system.

pointer: A variable containing the address of another variable, function, or constant.

pointer arithmetic: The use of addition or subtraction to change a pointer's value. Pointer arithmetic is typically used with array pointers, though it is not illegal on other kinds of pointers.

pointer-member operator: The –> operator, used with structure pointers to name a structure member.

pragma: An instruction to the compiler to perform an action at compile time.

precedence: The relative position of an operator in the hierarchy that determines the order in which expressions are evaluated.

preprocessor: A text processor that manipulates the contents of a C source file during the first phase of compilation.

preprocessor directive: See "directive."

prototype: See "function prototype."

recursion: The process by which a function calls itself.

register variable: An integer variable that is placed in a machine register, which may cause the program to be smaller and faster.

remapping: The process of assigning new color values to color indexes. Remapping a color index changes the screen color of any pixels that have been drawn with that color index.

reserved word: See "keyword."

return value: The value that a function returns to the calling function.

run time: The time during which a previously compiled and linked program is executing.

run-time library: A file containing the routines needed to implement certain functions of the Microsoft QuickC language.

scaling: The mapping of real-window coordinates to viewport coordinates.

scope: The parts of a program in which an item can be referenced by name. The scope of an item may be limited to the file, function, block, or function prototype in which it appears.

screen mode: See "video mode."

single precision: A real (floating-point) value that occupies four bytes of memory. Single-precision values are accurate to seven decimal places.

sizeof operator: A C operator that returns the amount of storage, in bytes, associated with an identifier or a type.

source file: A text file containing C language code.

standard error: The device to which a program sends its error messages unless the error output is redirected. In normal DOS operation, standard error is the display. The predefined stream **stderr** is associated with standard error in the C language.

standard input: The device from which a program reads its input unless the input is redirected. In normal DOS operation, standard input is the keyboard. The predefined stream **stdin** is associated with standard input in the C language.

standard output: The device to which a program sends its output unless the output is redirected. In normal DOS operation, standard output is the display. The predefined stream **stdout** is associated with standard output in the C language.

static variable: A variable that keeps its value even after the program exits the block in which the variable is declared.

stream: A sequence of bytes flowing into (input) or out of (output) a program.

stream functions: Run-time library functions that treat data files and data items as "streams" of individual characters.

string: An array of characters, terminated by a null character (**\0**).

string constant: A string of characters and escape sequences enclosed in double quotes (""). Every string constant is an array of elements of type **char**. See "character constant."

structure: A set of elements, which may be of different types, grouped under a single name.

structure member: One of the elements of a structure.

structure pointer: A pointer to a structure. Structure pointers identify structure members by specifying the name of the structure, the pointer-member operator (**–>**), and the member name.

symbolic constant: An identifier defined in a **#define** preprocessor directive to represent a constant value.

tag: The name assigned to a structure, union, or enumeration type.

ternary operator: An operator used in ternary (three-part) expressions. C has one ternary operator, the conditional operator (**? :**).

text: Ordinary, readable characters, including the uppercase and lowercase letters of the alphabet, the numerals 0–9, and punctuation marks.

text file: A file of ASCII characters that you can read with the TYPE command or a word processor.

text format: A method of disk storage in which all data are converted to ASCII format.

text mode: See "video mode."

text window: A window defined in row and column coordinates where text output to the screen will be displayed. Text printed beyond the edge of the text window is not visible. The default text window is the whole screen.

two's complement: A kind of base-2 notation used to represent positive and negative numbers in which negative values are formed by complementing all bits and adding 1 to the results.

type: A description of a set of values. For example, the type **char** comprises the 256 values in the ASCII character set.

type cast: An operation in which a value of one type is converted to a value of a different type.

type checking: An operation in which the compiler verifies that the operands of an operator are valid, or that the actual arguments in a function call are of the same types as the corresponding formal parameters in the function definition and function prototype.

type declaration: A declaration that defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type.

typedef declaration: A declaration that defines a shorter or more meaningful name for an existing C data type or for a user-defined data type. Names defined in a **typedef** declaration are often referred to as "typedefs."

typeface: The style of displayed text.

type name: The name of a data type. See "type."

type qualifier: The keywords **short**, **long**, **signed**, and **unsigned**, which modify a basic data type.

type size: A measure of the screen area occupied by individual characters in a font, typically specified in pixels.

unary expression: An expression consisting of a single operand preceded or followed by a unary operator.

unary operator: An operator that takes a single operand. Unary operators in the C language are the complement operators (– ~ !), indirection operator (*), increment (++) and decrement (– –) operators, address-of operator (&), and **sizeof** operator. The unary plus (+) operator is legal but has no effect.

union: A set of values of different types that occupy the same storage space.

vector-mapped font: A font in which each character is defined in terms of lines and arcs.

VGA (Video Graphics Array): Many users refer to the video subsystem integrated into the PS/2 Models 50, 60, and 80, as well as the IBM PS/2 Display Adapter, as the "VGA."

video adapter: A printed-circuit card that generates video output. Well-known IBM PC video adapters include the MDA, CGA, HGC, EGA, MCGA, and VGA Adapters.

video mode: An integer that specifies the resolution and other characteristics of video output. QuickC supports 17 different video modes, although some of them are available only with certain video adapters.

viewport: A clipping region in which the origin (0, 0) may be redefined. The initial origin of a viewport is the upper left corner.

viewport coordinates: The integer coordinate system defined by the programmer for a specific viewport. By default, the viewport-coordinate system has the origin (0, 0) at the upper left corner of the viewport, but this may be changed by a call to **_setvieworg**.

visibility: The parts of the program in which a particular variable or function can be referenced by name. An item has global visibility if it is visible in all source files constituting the program and local visibility if its use is restricted.

white-space character: A space, tab, line-feed, carriage-return, form-feed, vertical-tab, or newline character.

window: An imaginary rectangle on the screen where output takes place. See "text window" and "window coordinates."

window coordinates: The coordinate system defined by the programmer.

Index

D

E

F

G

H

I

K

L

M

T

MICROSOFT PRODUCT ASSISTANCE REQUEST

Microsoft Product Support Services - Phone (206) 454-2030

Instructions

When you need assistance with a Microsoft product, call our Product Support Services group at (206) 454-2030. So that we can answer your question as quickly as possible, please gather all information that applies to your problem. Note or print out any on-screen messages you get when the problem occurs. Have your manual and product disks close at hand and have all the information requested on this form available when you call.

Diagnosing a Problem

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

1. Can you reproduce the problem?
   ☐ yes ☐ no
2. Does the problem occur with another copy of the original disk of your Microsoft Software?
   ☐ yes ☐ no
3. Does the problem occur with another system (if available)?
   ☐ yes ☐ no
4. If you were running other windowing or memory-resident software at the same time, does the problem also occur when you don't use the other software?
   ☐ yes ☐ no

Product

____
Product name

____________ ____________
Version Number Registration Number

Software

Operating System

____
Name/Version number

Windowing Environment

If you were running Microsoft Windows or another windowing environment, give name and number of windowing software:

____

CD ROM Software

____
Name/Version number

Other Software

Name/Version number of any other software you were running when problem occurred, including memory-resident software (such as keyboard enhancers or print spoolers):

____

____

____

Hardware

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

Computer

______________________ ______________
Manufacturer/model Total memory

Floppy-disk drives
Number: ❒ 1 ❒ 2 ❒ Other
Size: ❒ 3 1/2" ❒ 5 1/4"
Number of Sides: ❒ 1 ❒ 2
Density: ❒ Single ❒ Double ❒ Quad
Capacity:
5 1/4": ❒ 160K ❒ 360K ❒ 1.2 megabytes
3 1/2": ❒ 360K ❒ 400K ❒ 720K ❒ 800K ❒ 1.4 megabytes

System Memory

______________________ ______________
Manufacturer/model Total memory

(If using DOS, you can run CHKDSK to determine the amount of memory available. If using Apple Macintosh Finder, select "About The Finder..." from the Apple menu to determine the amount of memory available.)

Peripherals

Hard Disk

______________________ ______________
Manufacturer/model Capacity(megabyte)

Printer/Plotter

____
Manufacturer/model

❒ Serial ❒ Parallel

Printer peripherals, such as font cartridges, downloadable fonts, sheet feeders:

____

____

Mouse
Microsoft Mouse: ❒ Bus ❒ Serial ❒ InPort™
❒ Other

____
Manufacturer/model

Boards
❒ Add-on RAM board

____
Manufacturer/model

❒ Graphics-adapter board

____
Manufacturer/model

❒ Other boards installed

____
Manufacturer/model

Modem

____
Manufacturer/model

CD ROM Player

____
Manufacturer/model

Version of Microsoft MS-DOS® CD ROM Extensions:

____

Network

Is your system part of a network? ❒ Yes ❒ No

____
Manufacturer/model

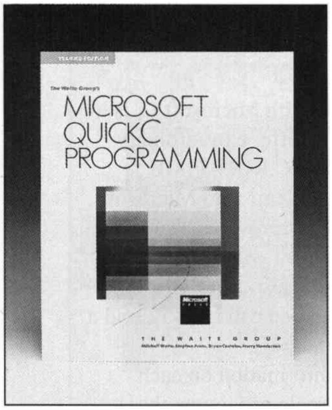
What hardware and software does your network use?

____

____

____

The Authorized Editions— Microsoft Press® Books



THE WAITE GROUP'S MICROSOFT® QUICKC® PROGRAMMING, 2nd ed.

The Waite Group

"This C book gets an A." ***Computer Currents***

Your springboard to the core of Microsoft QuickC. This book is loaded with practical information and advice on every QuickC element, along with hundreds of specially constructed listings. Included are the tools to help you

- master QuickC's built-in libraries
- manage file input and output
- work with strings, arrays, pointers structures, and unions
- use the graphics modes
- develop and link large C programs
- debug your source code

MICROSOFT QUICKC PROGRAMMING—the essential reference for every QuickC programmer has been updated to version 2.5 and covers the Microsoft Quick Assembler, the expanded graphics libraries, the customizable editor, and ANSI C compatibility.

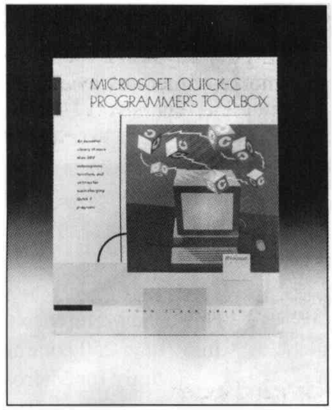
656 pages, softcover 7 3/8 x 9 1/4 $22.95 Order Code QCPR2 (available May 1990)

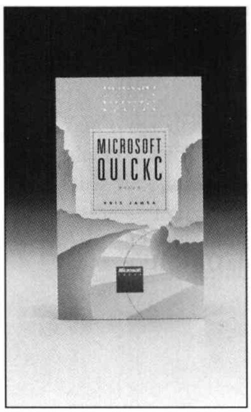
MICROSOFT® QUICKC® PROGRAMMER'S TOOLBOX

John Clark Craig

This is an essential collection of more than 200 programs, functions, and utilities designed to supercharge QuickC programs—a gold mine for novice and intermediate QuickC programmers. They offer solutions in modules that can be applied immediately—you save hours of development time, and you sharpen your QuickC programming in the process. The toolbox routines offer fine examples of structured programming. Included in the book are programs, functions, and utilities that enable you to ■ access, use, and control a mouse in creative ways ■ develop and customize menus ■ draw and fill geometric figures using QuickC's graphics functions ■ format text files for PostScript printing ■ access DOS and BIOS functions through software interrupts ■ work with fractions and complex numbers ■ manipulate strings in a variety of ways.

544 pages, softcover 7 3/8 x 9 1/4 $22.95 Order Code QCPRTO





MICROSOFT® QUICKC®: PROGRAMMER'S QUICK REFERENCE

Kris Jamsa

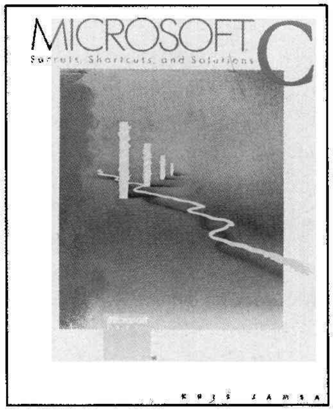
Whether you're new to Microsoft QuickC or a veteran, here's concise, handy information you'll want at your fingertips while you program. This reference is a great place to find instant refreshers and quick tutorials. In addition to providing a brief overview of QuickC—its related graphical interface, program lists, compiler restrictions, and pragmas—Jamsa covers: installing and starting QuickC, accessing QuickC Help, debugging your programs, developing large programs and libraries in QuickC, accessing the run-time library, and more. A final section includes a complete listing of the QuickC compiler error messages.

176 pages, softcover 4 3/4 x 8 $6.95 Order Code QRQC

Microsoft Press books are available wherever fine books are sold, or credit card orders can be placed by calling ***1-800-MSPRESS.***

Authoritative Microsoft Press® C Books

MICROSOFT® C: SECRETS, SHORTCUTS, AND SOLUTIONS



736 pages, softcover

7 3/8 x 9 1/4 $24.95

Order Code CSESH

Kris Jamsa

Here is a fact-filled resource for any current or aspiring Microsoft C programmer. Each chapter highlights specific C programming facts, tips, and traps so that key information or items of special interest are immediately accessible. If you're new to C, Microsoft C, or even Microsoft QuickC, you'll quickly master the fundamentals of the language with this book. Hundreds of short sample programs encourage experimentation. For experienced C programmers, advanced information covers

- accessing the DOS command line
- expanding wildcard characters into matching filenames
- using I/O redirection
- mastering dynamic memory allocation
- enhancing your program's video appearance
- using the MAKE and LIB tools

MICROSOFT C: SECRETS, SHORTCUTS, AND SOLUTIONS has the advanced information you need to hone your programming skills and make your Microsoft C programs faster, cleaner, and more efficient.

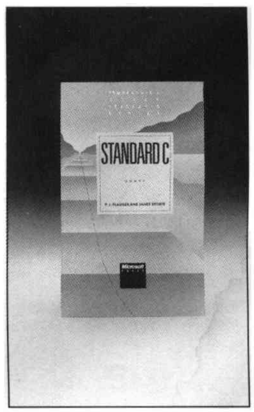
STANDARD C: PROGRAMMER'S QUICK REFERENCE

P. J. Plauger and Jim Brodie

At last! Here's all the information you need to read and write Standard C programs that conform to the recently approved ANSI and ISO standard for the C programming language. You'll find

- concise descriptions of all aspects of Standard C in this one-of-a-kind guide
- scores of diagrams that illustrate the syntax rules
- notes on writing portable C programs and converting older C programs to Standard C
- a complete listing of C's predefined names

You'll discover the most efficient—and effective—ways to read and write data between the program and data files and how you can use the formatting functions to simplify input and output. And there is concise information—along with brief descriptions—on all functions, macros, and types defined in the library. Keep this guide handy! Even if you're familiar with an earlier dialect of C, you'll refer to this guide quite frequently.



224 pages, softcover

4 3/4 x 8 $7.95

Order Code QRSTC

*Microsoft Press books are available wherever fine books are sold, or credit card orders can be placed by calling **1-800-MSPRESS.***

Documentation Feedback – QuickC® Version 2.5

Help us improve our documentation. After you've become familiar with our product, please complete and return this postage-paid mailer. Comments and suggestions become the property of Microsoft Corporation.

Which statement best describes your experience with C?

____ I haven't had much programming experience in any language.
____ I have used other languages, but I'm new to C.
____ I have used C occasionally, but I'm still unfamiliar with many of its features.
____ I use C regularly in my professional work, but I'm not a full-time programmer or developer.
____ I'm a full-time programmer or developer using C regularly.

How long ago did you buy this QuickC package?

____ months

Have you read *Up and Running* all the way through?

____ I haven't used it at all.
____ I've read part of it. Which parts? __________

____

____ I've read it all the way through.

Have you used the online training program, "Learning the Microsoft QuickC Environment"?

____ I haven't used it at all.
____ I've used part of it. Which parts? __________

____

____ I've followed it all the way through.

Which statement best summarizes your response to the C language information in *C for Yourself*?

____ It's too simple; I need more in-depth information.
____ It's about right; I can usually understand it without much difficulty.
____ It's too technical; I find it hard to read and apply.

Normally, what percentage of your programming is done

____ In the QuickC environment?
____ Outside the environment (compiling from the command line)?

If you normally program from outside the QuickC environment, using the utilities documented in *Tool Kit*, which part contains the information you need the most?

____ I find most material I need in Part 1, "Tool Kit Tutorial."
____ I find most material I need in Part 2, "Reference to QuickC Tools."
____ I use both parts about equally.
____ Doesn't apply to me.

In this QuickC package, some information is provided on line and some in book form. What's your opinion of this mix?

____ I wish more information were available on line. Please specify. __________

____

____

____ I wish more information were available in book form. Please specify. __________

____

____

____ I feel the balance is about right.

Were there any topics you felt weren't covered well enough anywhere in the documentation? Please explain. __________

____

____

____

Overall, how well does the QuickC documentation meet your needs? Rate each from 1 (does not meet your needs at all) to 5 (meets your needs perfectly).

____ *Up and Running* __________
____ *C for Yourself* __________
____ *Tool Kit* __________
____ QC Advisor (online help) __________

____

____ "Learning the QuickC Environment" (online tutorial) __________

Now, please return to the question above and tell us, in the space after each item, the main reason for your rating.

Use the back of this card for additional comments. Please note any errors and special strengths or weaknesses in areas such as programming examples, indexing, and overall organization. Which parts do you refer back to most frequently?

Name

Address

City/State/Zip

Phone ( ) (home) ( ) (work)

May we contact you for additional information about your comments? Yes ____ No ____

Additional comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.108 BELLEVUE, WA U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Languages—QuickC 2.5

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®
Making it all make sense™

0390 Part No. 10423